ADDING MACHINE INTELLIGENCE TO HYBRID MEMORY MANAGEMENT

A Dissertation
Presented to
The Academic Faculty

By

Thaleia Dimitra Doudali

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2021

ADDING MACHINE INTELLIGENCE TO HYBRID MEMORY MANAGEMENT

Thesis committee:

Professor Ada Gavrilovska, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Sudhanva Gurumurthi
Advanced Micro Devices Inc. (AMD)

Professor Alexey Tumanov
School of Computer Science
Georgia Institute of Technology

Date approved: July 21, 2021

To my parents, Elsa and Dimitris

To my brothers, Giannis and Stelios

ACKNOWLEDGMENTS

First and foremost I owe my deepest gratitude to my advisor Ada Gavrilovska. She gave me the time, resources and opportunities to grow as an independent researcher. I feel truly lucky and privileged to have such a fantastic advisor, this is something that should not to be taken for granted. Ada always believed in my potential and guided me through challenging times and milestones. I admire her on how she balances life, family and career. She was the key to me having an enjoyable PhD journey and pursuing a career in academia. Her support in my academic job search was essential and I look forward to having her as a life long mentor in career and life.

Next, I am very grateful to my committee members for their great support, feedback and excitement about my work. I really appreciate the enthusiastic interest in this thesis from Professors Tushar Krishna and Alexey Tumanov and I greatly admire their expertise and academic achievements. Next, I am very grateful to Dr. Sudhanva Gurumurthi for his mentorship and support in my academic job search. He was one of my mentors during a summer internship at AMD Research, that resulted in a very well received publication and fundamental contribution of this thesis. Last but not least, it has been a pleasure to interact with Professor Vivek Sarkar, the current chair of the School of Computer Science (SCS). I value his leadership and genuine interest in student well being. I am excited to see what he will achieve alongside the Graduate Student Association of SCS, that I helped put together and I will serve as an Alumni Outreach chair.

My time at Georgia Tech has been incredible and filled with so many memories and experiences. I thoroughly enjoyed volunteering and organizing activities for the School of Computer Science and interacting with the brilliant faculty and graduate students. I also served on the executive board of the Hellenic Society at Georgia Tech, bringing together the community of Greek students and faculty. I really liked attending conferences, presenting posters and papers, networking and traveling all over the world. I sincerely appreciate hav-

ing these opportunities, that were essential in developing an interest to pursue an academic career.

All of these experiences would not have been the same without my incredible friends. My wonderful labmates Carol, Harshit, Ranjan, Daniel, Misun, Jin, Jim, Tony and Pradeep have been the greatest set of friends and collaborators, I wish them best of luck and success. To my incredible international friends Samira, Sajad, and Saurabh, I will always remember so many fun moments. Last but surely not least, I cherish my Greek friends Ria, Eva, Kyveli, Alexandros, Alexandra and Thanos. Thank you for all of the laughs and love, especially throughout the Covid-19 pandemic, I will dearly miss you and hope we soon end up closer geographically.

Finally, this thesis is dedicated to my parents and two older brothers, for their never-ending love, support and guidance. Cheers to 3/3 Dr. Doudalis in computer science!

This thesis has been partially funded by the Department of Energy UNITY project under the SSIO program, the Department of Energy SICM project under the ECP program, the NSF awards SPX-1822972 and CNS-2016701, a Facebook research award, and by Intel's VLAB program that provided access to persistent memory hardware.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

APU Accelerated Processing Unit

CXL Compute Express Link

DNN Deep Neural Network

DRAM Dynamic Random Access Memory

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HPC High Performance Computing

LRU Least Recently Used

LSTM Long Short Term Memory

MI Machine Intelligence

ML Machine Learning

NUMA Non Uniform Memory Access

NVM Non Volatile Memory

PMEM Persistent Memory

RNN Recurrent Neural Network

TPU Tensor Processing Unit

SUMMARY

Computing platforms increasingly incorporate heterogeneous memory hardware technologies, as a way to scale application performance, memory capacities and achieve cost effectiveness. However, this heterogeneity, along with the greater irregularity in the behavior of emerging workloads, render existing hybrid memory management approaches ineffective, calling for more intelligent methods. To this end, this thesis reveals new insights, develops novel methods and contributes system-level mechanisms towards the practical integration of machine learning into hybrid memory management, boosting application performance and system resource efficiency. The specific contributions of this thesis are as follows.

First, this thesis builds Kleio, a hybrid memory page scheduler with machine intelligence. Kleio deploys Recurrent Neural Networks to learn memory access patterns at a page granularity and improve upon the selection of dynamic page migrations across the memory hardware components. Kleio cleverly focuses the machine learning on the page subset whose timely movement will reveal most application performance improvement, while preserving history-based lightweight management for the rest of the pages. In this way, Kleio bridges on average 80% of the relative existing performance gap, while laying the grounds for practical machine intelligent data management with manageable learning overheads.

Second, this thesis contributes Cori, a system-level solution for tuning the operational frequency of periodic page schedulers for hybrid memories. Cori synthesizes information on data reuse to properly identify the data movement frequencies to be tested, reducing by 5$\times$ the number of tuning trials compared to existing empirical or insight-less tuning approaches. In this way, Cori delivers application performance levels within 3% from the case of optimally selected frequency, eliminating the 10%-100% performance gap created when using frequencies currently adopted by related works. Such improvements are complimen-

tary to the use of machine learning and further boost its effect on application performance.

Third, this thesis contributes Coeus, a page grouping mechanism for hybrid memory page schedulers with machine intelligence, such as Kleio. Coeus leverages the data reuse insights revealed by Cori to fine-tune the granularity at which patterns are interpreted by the page scheduler, increasing the pattern similarity across pages. Then, Coeus groups together the pages that share the same access behavior, enabling the training of a single Recurrent Neural Network per page cluster. As a result, Coeus reduces by almost $3\times$ the associated learning overheads compared to Kleio. In addition, Coeus achieves $3\times$ higher application performance, by the combined effects of applying machine learning to more pages and by performing management operations at a fine-tuned granularity.

Finally, this thesis contributes Cronus, an image-based page selector for hybrid memory page schedulers with machine intelligence, such as Kleio. Cronus uses visualization to accelerate the process of selecting which page patterns should be managed with machine learning. The visualization reveals spatial and temporal correlations across pages, that Kleio cannot capture during its page selection process. Instead, Kleio uses elaborate performance estimation models that come with non trivial operational overheads. Cronus builds a lightweight visualization pipeline that detects page access patterns for machine learning-based management. The quality of the selected pages is comparable to Kleio's and delivers similar levels of application performance, in return for $75\times$ reduction in the selection time. Cronus lays the foundations for future use of visualization and computer vision methods in memory management, such as image-based memory access pattern classification, recognition and prediction.

CHAPTER 1
INTRODUCTION

Heterogeneous hardware emerged to address the slowdown of Moore's Law and the exponentially growing demand for compute and storage resources by popular big data analytics, applications of artificial intelligence and scientific simulations. Regarding the memory substrate, non volatile memory technologies of massive capacity capabilities emerged to enable fast data retrieval and storage for data-intensive workloads, in response to the scaling limitations and skyrocketing cost of the Dynamic Random Access Memory (DRAM). For instance, Intel's Optane DC persistent memory [1] provides terabytes of memory at 1/3 of DRAM's cost [2]. Intel offsets the at least $3\times$ slower access speeds of persistent memory [3] by packaging together gigabytes of DRAM, which account for as little as 6% of the platform overall memory capacity, thus creating a hybrid memory environment.

The efficient resource management of hybrid memory, via proper data tiering, allows for the desired performance levels of the aforementioned classes of applications. To achieve this, well established approaches in hybrid memory management build the necessary mechanisms to maximize the utility of the fastest available memory component via corresponding dynamic movement of frequently accessed data. The task to identify which data is most appropriate to move and at what times, is particularly challenging, depending on the available data access information and performance estimates. Current solutions span the software stack from algorithm- [4, 5], profiling- [6, 7], library- [8], runtime- [9, 10, 11] to operating system-level [12, 13, 14, 15, 8, 16] solutions. Custom APIs [9, 6, 17, 8] and specialized hardware [13, 12, 18, 19, 15, 20] are frequently proposed as part of the solutions across the software stack, to collect the necessary data access information and deliver the desired performance levels.

However, the ever increasing complexity of emerging workloads and of the system pa-

rameter configuration space, breaks the effectiveness of current system-level and application-agnostic solutions. More specifically, the use of current heuristics, that are fine-tuned for conventional workloads, is not effective for analytics with more intricate or random accesses. In addition, the effectiveness of heavily used empirical configurations is vulnerable to such emerging workloads, due to the impermissible overheads of fine-tuning all possible combinations of parameters and scenarios. Therefore, existing approaches are not robust across classes of emerging workloads and configurations of the heterogeneous hardware. This results in substantial loss in performance and resource efficiency of the heterogeneous memory hardware.

The increased complexity and resulting performance gap call for more intelligent and insight-based solutions compared to existing human-derived heuristics and settings. Yet, system-level solutions need to be lightweight so as to be commercially adopted and to maximize performance with minimal software-level overheads. Although the effectiveness of machine intelligence, that is machine learning methods, in addressing complexity is very appealing for the purpose of hybrid memory management, its system-level adoption needs to be sophisticated, due to the non-trivial learning overheads that are associated with the massive memory footprints of modern data analytics. This raises a number of questions: *How to practically integrate machine learning in the system-level resource management? At which part of the memory management software stack? Which machine learning method to use and what exactly to predict? How to amortize the training costs given the massive application data sizes? Is machine learning sufficient to maximize application performance and system resource efficiency?* We next discuss, in more detail, the need for machine intelligence and the challenges introduced by its system-level integration.

1.1 Statement of Problem

Emerging complex application classes break the effectiveness of conventional hybrid memory management approaches. The growing adoption of machine learning methods

across application domains creates a new class of workloads that require adequate system-level support [21, 22, 23, 24, 25, 26]. Similarly, scientific simulations now capture even more complex phenomena and relations [27]. These applications and methods have vastly more intricate execution phases and access patterns, than traditional analytics. Yet, hybrid memory management approaches have not evolved to address this newfound complexity in data access behaviors, and continue using heuristics and approaches that are only proven to work well across classes of conventional workloads.

In more detail, the use of access history information to predict future access behaviors has been effective for traditional classes of workloads with sequential, strided and regular access patterns. However, it generates vastly inaccurate predictions when reacting to sudden changes in execution phases or to completely irregular behavior, which is predominant in emerging workloads. This results in an inefficient selection of data to be moved across the memory components, due to mispredictions of upcoming patterns. In consequence, application performance degrades due to the suboptimal data tiering, as well as the wasteful resource utilization that delivers such data migrations. The created performance gap is substantial and requires novel access pattern prediction mechanisms enriched with machine intelligence to realize the full potential of the heterogeneous hardware.

The increased complexity in the parameter configuration space hinders the fine-tuning of critical operational parameters. The large configuration space of heterogeneous hardware, the intricate resource requirements of emerging workloads and the explosion of solution-level parameters, vastly complicate the configuration space. This leads to the empirical setting of certain knobs, in an effort to minimize the associated tuning overheads. For instance, the frequency of data movements over hybrid memory has always been empirically set at certain fixed values, that surprisingly varies substantially across related works [12, 14, 28, 13, 15]. In consequence, such settings are not robust to new workloads classes or scenarios that were not included in their experimental evaluation.

However, the importance of this parameter is enormous since its operation is on the critical path of hybrid memory management, where a misconfigured value can lead to tremendous aggregate movement overheads. An insight-based tuning of this important parameter that enables minimal tuning effort, will be more effective and practical than current insight-less settings, and can deliver the full benefits from the heterogeneous hardware.

Performance improvements are currently feasible either with very specialized solutions or in return for significant operational overheads. To address the new requirements and behaviors of emerging workloads, researchers develop solutions with more intricate performance models [10, 29] and heavy profiling [6] that increase the management decision overhead, rely on new hardware support [13, 12, 29, 19], or resort in application-guided [6, 8] or runtime-specific [9, 10, 11] solutions. Yet, commercial system-level support for emerging hardware chooses practicality over robustness and effectiveness across applications. System-level solutions need to be lightweight and responsive to adhere to decision time guarantees and to seamlessly cooperate with other system-level components. For instance, to offer support for new technologies such as Intel Optane DC PMEM [1], Linux simply extended its well established and lightweight `autonuma` component, that was built for Non Uniform Memory Access (NUMA) platforms, to enable identification and migration of hot and cold pages across DRAM and persistent memory [30].

Therefore, it is essential to preserve the practicality guarantee, while delivering more intelligent hybrid memory management to boost application performance and system resource efficiency. Although machine learning can alleviate the human effort in this complex management space, its insight-less adoption in the decision pipeline comes with unacceptable learning overheads. This is inherent to the enormous problem size of system-level hybrid memory management, due to the massive memory footprints of emerging workloads executing over hybrid memory. Therefore, the use of machine learning needs to be judicious and sophisticated, minimizing not only the training and learning but also the

operational overheads of its system-level integration.

1.2 Thesis Statement

To realize practical resource management methods that use machine intelligence to maximize the performance benefits from and efficiency of emerging heterogeneous memory hardware, new solutions are needed that extend existing lightweight system-level techniques, and augment them with machine learning via new mechanisms for extracting insights about applications' memory access behaviors.

1.3 Contributions

In support of this statement, this thesis makes the following contributions.

Foundations for practical machine learning-based hybrid memory management. To address the inefficiency of existing approaches to accurately predict the complex access behavior of emerging workloads, this thesis introduces machine intelligence into the hybrid memory management. It proposes Kleio, a hybrid memory page scheduler with machine intelligence [31]. Kleio deploys Recurrent Neural Networks (RNNs) to learn memory access patterns, that existing history-based solutions fail to accurately predict. Kleio does so at the granularity of individual pages, to learn the pattern of their access frequency across periods of time. The novelty in the design of Kleio comes from the selection of a small page subset, whose machine intelligent management reveals most of the application performance improvement, while using existing history-based management for majority of the pages. The resulting hybrid management approach lays the ground for the practical integration of machine intelligence in the management of complex systems with heterogeneous memories. Kleio's impact is extremely promising, since it bridges on average 80% and up to 95% of the existing performance gap.

Fine-tuning critical hybrid memory management operations with data reuse insights.

To address the inefficiency and inconsistency of empirically configured data movement frequencies proposed across related works and for different classes of application, this thesis builds a tuning solution based on insights that can be captured at the system level. In more detail, it proposes Cori, a system-level solution for tuning the operational frequency of periodic page schedulers for hybrid memories [32]. The novelty of Cori comes from observations on data reuse times and their alignment with the data movement frequency. Cori synthesizes information on data reuse to properly identify the data movement frequencies to be tested, reducing by $5\times$ the number of tuning trials compared to existing empirical or insight-less tuning approaches, and realizing almost maximum possible application performance levels, within only a trivial margin of 3% on average from the case of optimally selected frequency. Thus, Cori enables the proper fine-tuning that enables maximum performance improvements that are suppplementary to the ones deriving from machine intelligent management. Importantly, Cori is effective across application classes, platform characteristics, and system-level memory management policies.

Scaling machine learning-based hybrid memory management with pattern clustering. To address the significant training overheads when introducing machine learning into hybrid memory management, due to the huge memory footprints of target workloads, this thesis builds a clustering mechanism to further reduce the associated learning overheads. More specificaly, it builds Coeus, a page grouping mechanism that enables machine intelligent page schedulers to train, in parallel, different models per large page clusters. We identify that the resource and time requirements for training machine learning models vary up to $9\times$ across workload classes and input sizes. To reduce these requirements, Coeus leverages the data reuse insights revealed by Cori to fine-tune the granularity at which patterns are interpreted by the page scheduler, increasing the pattern similarity across pages. Then, Coeus groups together the pages that share the same access behavior, enabling the training of a single Recurrent Neural Network per page cluster. As a result, Coeus reduces

by almost $3\times$ the associated learning overheads compared to Kleio. In addition, Coeus achieves $3\times$ higher application performance, by the combined effects of applying machine learning to more pages and by performing management operations at a fine-tuned granularity.

Reducing the operational overheads of machine learning-based hybrid memory management with pattern visualization. To address the non trivial operational overheads when selecting the pages whose machine learning-based management would benefit application performance, this thesis proposes a lightweight approach based on analysis of visualized representations of memory access traces. To this end, this thesis contributes Cronus, an image-based page selector for hybrid memory page schedulers with machine intelligence, such as Kleio. Cronus uses visualization to accelerate the process of selecting which page patterns should be managed with machine learning. The visualization reveals spatial and temporal correlations across pages, that Kleio fails to capture since it focuses the analysis on a per-page basis. Instead, Kleio uses elaborate performance estimate models that come with non trivial operational overheads. Cronus builds a lightweight visualization pipeline that detects page access patterns for machine learning-based management. The quality of the selected pages is comparable to Kleio's and delivers similar levels of application performance, in return for $75\times$ reduction in the selection time. Cronus lays the foundations for future use of visualization and computer vision methods in memory management, such as image-based memory access pattern classification, recognition and prediction.

1.4 Organization

The remainder of this thesis document is organized as follows. Chapter 2 includes a background description of established methods and current challenges in hybrid memory management. In addition, the chapter highlights the gap in application performance left by

existing management solutions, that this thesis identifies and which motivates the thesis contributions.

Chapter 3 describes the experimental methodology that majority of the thesis follows. The thesis contributes a lightweight hybrid memory simulation and performance estimation model that is validated over a native hardware platform.

Chapter 4 describes which machine learning method to use and at which part of the hybrid memory management stack to be integrated. The chapter presents and evaluates Kleio, a system-level machine learning-based solution to manage hybrid memory.

Chapter 5 analyzes the effects on performance when managing hybrid memory at frequencies higher and lower than the application data reuse times. Then, the chapter describes how to incorporate the revealed insights into a system-level frequency tuning solution, Cori, that maximizes application performance.

Chapter 6 explores and identifies the limited practically and effectiveness of using well-established data clustering methods to reduce the learning overheads of machine intelligent hybrid memory managers. Instead, it leverages the data reuse insights revealed earlier in the thesis and presents Coeus, a lightweight page grouping mechanism to identify page patterns for machine learning.

Chapter 7 leverages visualization to better understand the correlations across pages whose machine learning-based management benefits performance. The chapter describes how to integrate visualization and computer vision techniques to identify pages for machine learning, presents and evaluates the proposed solution Cronus.

Chapter 8 provides a brief survey of related work, and Chapter 9 summarizes the contributions and lessons learned in this thesis. The thesis concludes with some thoughts on future directions, with respect to the practical use of machine learning and the integration of computer vision techniques in system-level resource management.

CHAPTER 2

MOTIVATION

This chapter provides background information on emerging heterogeneous hardware technologies, their commercial availability and deployment in datacenter and exascale computing environments. In addition, it highlights the increased complexity in the configuration of the hybrid memory in current systems and the data access behaviors of emerging workloads. Finally, this chapter includes experimental results that motivate the need for more intelligent hybrid memory management solutions, due to the significant gap in application performance left by existing system approaches and configurations.

2.1 Background

2.1.1 Emerging Hybrid Memory Platforms

In the current post-Moore era of computing, the traditional model of homogeneous DRAM-only memory systems is replaced with heterogeneous hardware to massively scale the main memory capacity under permissible system cost. Figure 2.1 summarizes the recent trends in hybrid memory configurations that include emerging hardware technologies and resource disaggregation techniques. New hardware technologies introduce different trade-offs of cost, speed, capacity and capabilities such as programmability and data persistence. For instance, the Non Volatile Memory (NVM) hardware released by Intel, that is the Optane DC Persistent Memory (PMEM) [1] provides terabytes of persistent data storage, at around 3$\times$ times cheaper [2] than DRAM, in return for at least 3$\times$ slower access speed [3, 33]. Yet, with appropriate data management, the 375 gigabytes of available DRAM, packaged together with 6 terabytes of PMEM, are sufficient to deliver DRAM-like levels of application performance [3]. Similarly, High Bandwidth Memory (HBM) deliver 5$\times$ - 10$\times$

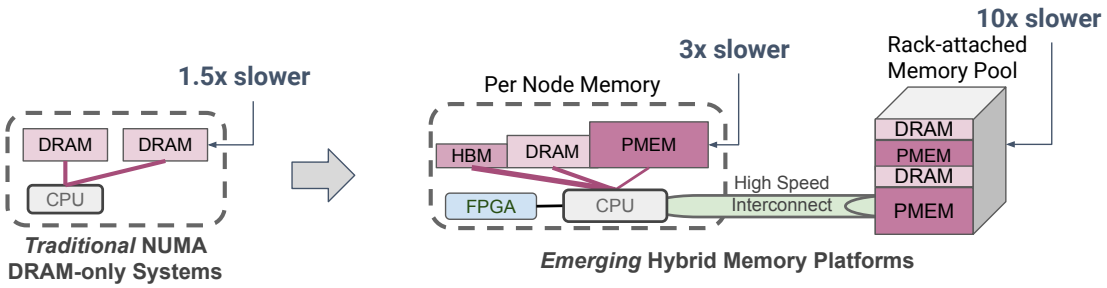


Figure 2.1: Heterogeneous memory hardware and memory disaggregation provide massive memory capacities in return for bigger disparity in the access speeds and configuration of the memory substrate, compared to traditional homogeneous systems.

more bandwidth compared to DRAM technologies [34]. It is widely used in the TOP500 Supercomputers at capacity of tens of gigabytes per node, either standalone [35] or together with DRAM [36]. Finally, well established techniques of resource disaggregation [37, 38], now aggregate both volatile and non volatile node-local memory resources into a massive shared pool of remotely accessible hybrid memory, and deliver high-speed data transfers with novel interconnection fabrics and standards like Gen-Z[39] and Compute Express Link (CXL)[40]. With these different hardware technologies we transition into the exascale era of compute with platforms that exhibit extreme heterogeneity, since they can include deep and wide hierarchies of hybrid memory and storage components, accelerator-near memories, new interconnection fabrics and resource disaggregation. Managing intelligently such complex computing platforms is most challenging and necessary than ever before across high performance computing [41, 42] and datacenter environments [43, 44, 45].

2.1.2 Hybrid Memory Organization

The increased number of distinct memory technologies in the main memory substrate adds new dimensions in the complexity of their configuration. Each hybrid memory unit may exhibit different properties in terms of speed, capacity, programmability or other parameters, complicating the decision regarding their optimal configuration. In addition, the available

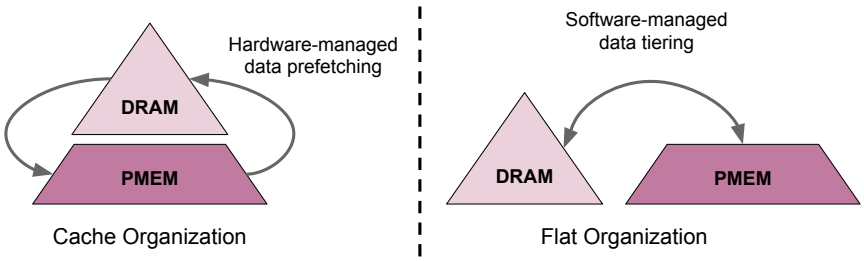


Figure 2.2: The cache organization of hybrid memory enables hardware-managed data prefetching techniques from PMEM to DRAM. The flat organization of hybrid memory allows for software-based control of the data tiering across DRAM and PMEM.

memory components can be organized in two primary ways or in a combination thereof. As depicted in Figure 2.2, for the example case of two memory tiers, there is the vertical and horizontal hybrid memory organization. In the vertical (otherwise cache) organization, one memory unit acts as a cache for the other and is managed by the hardware. In the horizontal (otherwise flat) organization, all memories 'lay flat' and are managed by software – the operating system or applications themselves. For instance, these correspond to the *Memory* and *App-direct* modes in Intel's Optane DC PMEM platform [1]. Each organizational mode introduces different trade-offs with respect to system resource efficiency and application performance. For instance, recent work has shown that the cache organization improves performance of graph applications [46]. In contrast, the flat organization allows for lower energy cost and higher bandwidth use [47, 33], and a number of hardware and software techniques have recently been proposed to further improve the associated management overheads [13, 12, 18, 19, 15]. This thesis is primarily focused on this 'flat mode' configuration of heterogeneous memories, that is explicitly managed by the systems software.

2.1.3 Emerging Complex Workloads

New classes of popular workloads include machine learning methods with complex matrix operations, massive graphs of random connectivity and scientific simulations that capture irregular behaviors and phenomena. Figure 2.3 captures examples of such access patterns

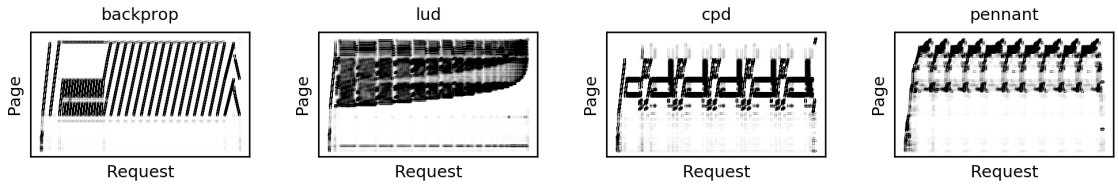


Figure 2.3: Memory access patterns of emerging workloads with increasing complexity in data access behavior. From sequential strides (`backprop`), triangular traversals (`lud`), sparse tensors (`cpd`) to iterations of random accesses (`pennant`).

in increasing order of complexity. New or extended benchmark suites [21, 22, 23, 24, 27, 48] introduce such emerging workloads and together with well established suites [25, 26, 49, 50] drastically augment the application classes that require robust support for high performance over heterogeneous hardware in particular. In particular, aritificial intelligence is deployed in so many domains and use cases, that new systems are being design to specifically optimize such workloads and pipelines [51, 52].

Emerging workloads add complexity to the hybrid memory management because it is harder to predict the randomness, frequent phase changes and other irregularities in their access behaviors, compared to conventional workloads. The effectiveness of hybrid memory tiering relies on accurately forseeing the upcoming trends in access patterns, so as to timely migrate future frequently accessed data in the fastest memory unit. This is particularly challenging given the use of past access history, that is readily available, since it is not always sufficient to predict new patterns or old trends outside the retained information. This results in a selection of data movements that do not optimize the data tiering, reduce the utility of the fastest memory and waste resources with non useful migrations. This translates in degradation of performance and efficiency. Therefore, such emerging classes of applications require hybrid memory management approaches with more predictive capabilities and fine-grained access pattern predictions.

2.2 Performance Gap

The previous section describes all the factors that increase the complexity of hybrid memory management, from the hardware configurations all the way to emerging application domains. In this section, we summarize current solutions used in commercial systems and capture the extent to which they are effective. We reveal a significant gap in performance that can be bridged via more intelligent and insightful management decisions.

Overview of System-level Hybrid Memory Management. Well established operating system-level approaches for hybrid memory management, include a *page scheduler* component, that periodically monitors page access behavior and selects pages to migrate across the memory units. The selection of which pages to move is based upon a certain page scheduling *policy*, that aims to dynamically adjust the page placement across hybrid memory adapting to changes in the application's memory access patterns, to boost application performance and system resource efficiency.

Current state-of-the-art solutions leverage existing NUMA-based page migration support [12, 14, 15, 20, 28], or appropriately extend NUMA-based data balancing policies [16, 53]. The policies proposed vary depending on the available page access information and custom thresholds and heuristics that they use. The common factor is to move frequently accessed (hot) pages to faster memory technologies replacing cold ones, to accelerate the run time a workload spends in accessing memory. A policy is effective when it uses robust models to predict how hot or cold pages will be in the future, given a window of observed past access history. The *frequency* at which pages are monitored and moved is most commonly determined empirically, such that it offsets the time and resource cost of moving pages with the performance benefit from the improved page placement.

2.2.1 Hybrid Memory Management Policy

To motivate the need for improved hybrid memory management policies, we capture the gap in performance left by current history-based page scheduling policies, since the use of purely historical information is not robust towards sudden changes or randomness in page access patterns. We therefore assume an oracle page scheduler, that has a-priori knowledge of the workload's memory accesses, thus will always make an accurate page hotness prediction. Figure 2.4 shows the reduction in speedup of a history page scheduler, compared to an oracle one as a baseline, across a variety of hybrid memory capacity ratio configurations. We observe up to 50% performance degradation, compared to oracle page schedulers, for configurations that correspond to recent platforms, such as Intel's Optane with 1/16 PMEM to DRAM. This performance gap is due to the limited capabilities of accurately predicting future behaviors with purely historical information, that translates to a poor selection of page migrations by the page scheduler, wastes resources, reduces the efficiency of data tiering and ultimately hurts application performance. More detailed analysis follows in Section 4.2. Therefore, there is an opportunity to bridge this performance gap by building more robust prediction models that will forecast page access hotness with higher accuracy, resulting in page migrations that benefit performance.

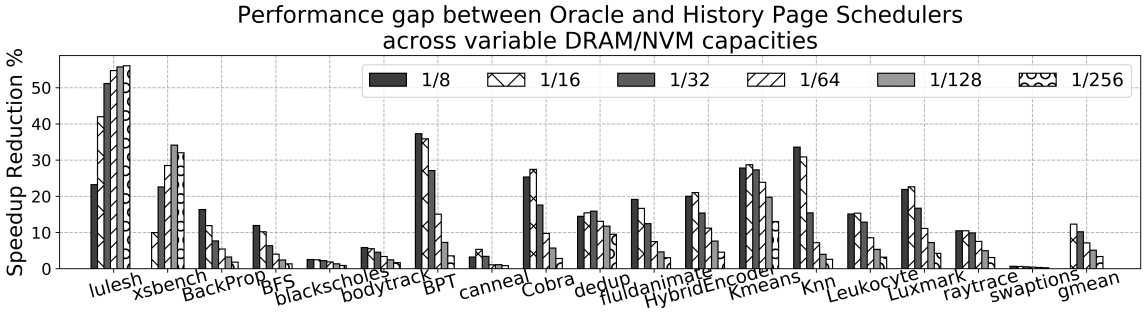


Figure 2.4: Performance gap due to sub-optimal data movement selection as a result of history-based access pattern predictions.

2.2.2 Hybrid Memory Management Frequency

Another factor that reveals a missed opportunity in the performance benefits from hybrid memory corresponds to current approaches in selecting the frequency at which page schedulers operate. The growing size of the configuration space of heterogeneous hardware and the substantial overheads of properly setting each parameter, leads to *empirical* tuning of the operational frequency of periodic hybrid memory management solutions. Suprisingly, the values selected by related works, as summarized in Table 2.1, vary within orders of magnitude, hinting towards potential ineffectiveness of these values for application classes and configurations not included in their tuning process.

To establish the effect on performance, we compare workload runtime when the page scheduler is configured with the proposed values versus when operating at the 'best' frequency that maximizes performance, as determined after extensive experimentation. To distinguish the effect that the page scheduling policy has on performance, we assume two policies similar to the ones described in the previous section. A reactive page scheduler operates similarly to a history one, and a predictive scheduler to the previously described oracle, as we further explain in Section 3.3.

Figure 2.5 reveals 10% - 80% performance degradation compared to the performance achievable with a best-case frequency, on average, across application domains and page scheduling policies. In Section 5.2 we also depict the effect on resource efficiency via the corresponding amount of data moved. Overall, we observe that no single proposed value works best across all applications and page schedulers. Therefore, there is an opportunity to close this performance gap with an insight-based tuning approach, rather than insight-less empirical procedures.

Table 2.1: Operational frequency of existing data tiering solutions.

Solution	Period Duration
Thermostat [12]	10 sec
Nimble [14]	5 sec
Ingens [28]	2 sec
HMA [13]	1 sec
Hetero-OS [15]	0.1 sec

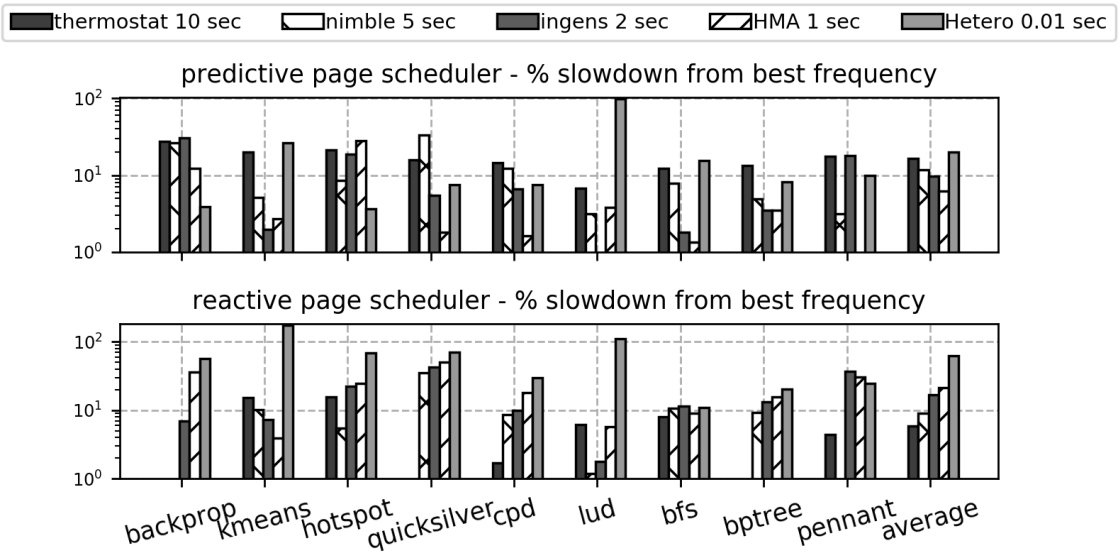


Figure 2.5: Performance gap due to sub-optimal data movement frequency selection as a result of empirical tuning of predictive and reactive (history-based) page schedulers.

2.3 Chapter Summary

This chapter summarized the latest hardware technologies, system-level solutions and challenges in hybrid memory management. In particular, this thesis identifies a significant gap in application performance left by current solutions that have limited capabilities to predict memory access behaviors and neglect to fine-tune critical operational parameters. The remainder of this thesis shows how to close this performance gap by building lightweight, practical and effective system components.

CHAPTER 3

HYBRID MEMORY SIMULATION AND PERFORMANCE MODELING

This chapter summarizes the experimental methodology followed throughout the thesis. To allow for lightweight exploration of the effect of applying machine learning in system-level hybrid memory management, this thesis contributes an open-source simulation environment [1]. In addition, we provide a public dataset [2] that includes memory access traces of benchmarks across application domains. Next, we describe in detail the simulation infrastructure and provide evidence on its performance estimate validation against workload execution over a native hardware hybrid memory platform.

3.1 Memory Access Trace Collection

Table 3.1 summarizes the applications that we selected for experimental evaluation from the Rodinia [26], Coral-2 [27] and ParTI! [23] benchmark suites. The selected benchmarks and mini-apps cover a wide range of application domains and memory access patterns. We use Intel's Pin [54] dynamic binary instrumentation tool to capture the memory address of the last level cache misses out of a simulated three level data cache hierarchy. In order to allow for reasonable trace sizes and analysis times we simulate a cache hierarchy of smaller but proportional capacity ratio to a native hybrid memory platform with Intel Optane DC Persistent Memory Modules (PMEM) [1], which contains 375 GB of DRAM and 6 TB of persistent memory. Then we fix the application data inputs such that we observe similar last level cache miss rate to application execution in the native hardware platform.

[1] https://github.com/GTkernel/cori-sim.git
[2] https://github.com/GTkernel/cori-sim/tree/master/traces

Table 3.1: Application kernels used in experiments.

Application	Kernel	Suite	Domain
Back Propagation	`backprop`	Rodinia	Machine Learning
Kmeans	`kmeans`	Rodinia	Machine Learning
HotSpot	`hotspot`	Rodinia	Physics Simulation
LU Decomposition	`lud`	Rodinia	Linear Algebra
Breadth-First	`bfs`	Rodinia	Graph Algorithms
B+Tree	`bptree`	Rodinia	Databases
Pennant	`pennant`	Coral-2	Hydrodynamics
Quicksilver	`quicksilver`	Coral-2	Monte-Carlo
CP Decomposition	`cpd`	ParTI!	Sparse Tensors

3.2 Hybrid Memory System Simulation

We develop a Python-based simulation environment that allows fast trace-based analysis similar to [13]. In particular, we assume a flat organization of fast (e.g., DRAM) and slow (e.g., PMEM) memory, similar to the App Direct mode configuration of the Intel Optane platform. Following the observed PMEM access speeds [3] we set a 1:3 latency and 1:0.37 bandwidth ratio between the fast and slow memory. We assume that the overall capacity of the memory system is equal to an application's memory footprint, split into 20% DRAM and 80% PMEM across all experiments. Since we are not using cycle-accurate simulation, we assume that a period is the time duration when a fixed number of memory requests are issued, e.g., 1,000 requests per period. To estimate the runtime we aggregate the access latency of the memory requests for their coresponding memory allocation across periods. In addition, we account for any limited bandwidth availability, by injecting appropriate delays given the number of memory requests serviced over a window of time. Finally, we add constant delays for every page migration and start of a period to account for the overhead of the page scheduler itself, using the proposed values in [13, 18].

3.3 Page Scheduling Policies

We extend the Python-based simulation with a page scheduler that periodically aggregates per page access counts from the collected access trace and migrates pages between fast and slow memory. The initial page allocation is done in an interleaved manner across memories, which is typical for NUMA systems. Every period the page scheduler identifies the pages that are frequently accessed (hot) and moves to fast memory any hot pages that reside in slow memory, replacing any Least Recently Used (LRU) pages. The number of page migrations per period is capped by the available fast memory capacity, since hot and LRU pages are swapped across hybrid memory. These page swaps happen asynchronously, assuming DMA support, and sequentially in order of (hot, LRU) page pairs.

We refer to this type of page scheduler, that makes a selection of page migrations using access history, as a **history / reactive** page scheduler, since it 'reacts' to the changes in the memory access pattern, as also done in [12, 14, 28, 13, 20, 15]. We also simulate an **oracle / predictive** page scheduler, that predicts memory access patterns, by having a-priori knowledge of the access pattern, described as the oracular baseline in [13]. The reactive page scheduler is configured to act upon a single period of past access history, and similarly the predictive page scheduler to make an access pattern prediction for the upcoming period.

3.4 Native Hardware Validation

We validate the accuracy of the application performance estimate that the aforementioned trace-based simulation generates, against workload execution over a native hybrid memory hardware platform. Next we summarize the internal functionality of the validation system and the experimental evidence is provided in the evaluation part of Chapter 5.

We have access to a server with Intel Optane DC Persistent Memory Modules (PMEM), which we configure in *App Direct* mode. The machine contains 375 GB of DRAM and 6

TB of PMEM. More specifically, we make use of a page migration module[3] built for Linux kernel version 5.4 that attaches to a target process and periodically selects 4 KB pages to move between DRAM and PMEM. Every period, the module identifies page accesses using the available OS-level information, as also done in [20, 15]. In more detail, the module determines which pages were accessed by scanning the target's page table entries and recording whether or not each accessed bit was set during that period. All accessed bits are then cleared so that they can be tested again during the next scan. To estimate the page hotness, the module calculates the exponential moving average (with a certain smoothing factor) of the page's accessed bit history and compare it with a hotness threshold that classifies a page as hot or cold, as also done in [28]. Then, utilizing the `move_pages()` function from the kernel's NUMA-based migration API, it asynchronously moves hot pages to DRAM and cold pages to PMEM. The kernel module dynamically adjusts the page migration cutoff, dividing the process memory footprint across DRAM and PMEM at a certain capacity ratio, that is 20% DRAM and 80% PMEM.

3.5 Chapter Summary

This chapter described the software and methodology that this thesis develops to design and evaluate its system-level contributions. The thesis builds a lightweight simulation environment that configures hybrid memory platform characteristics, captures system-level policies for data management and produces application runtime estimations. The accuracy of these estimates is validated against workload execution over a native hardware hybrid memory platform.

[3] https://github.com/GTkernel/x86-Linux-Page-Scheduler.git

CHAPTER 4
FOUNDATIONS FOR PRACTICAL MACHINE LEARNING-BASED MANAGEMENT

The previous chapters describe how existing hybrid memory management systems fail to maximize application performance and system resource efficiency, creating a significant gap in the attainable performance. This is due to the limited effectiveness of current approaches against the increased complexity of emerging workloads and platform configurations. To this end, this chapter introduces a Machine Learning (ML)-based hybrid memory management system, **Kleio**[1] [31], that improves performance via robust memory access pattern predictions. More importantly, this chapter proposes design foundations that lay the grounds toward the practical integration of machine learning inside system-level resource management. The chapter focuses on exploring which machine learning method is most practical to use, at what part of the hybrid memory management software stack and how to enable high prediction accuracy in return for permissible training overheads, due to the massive data sizes of modern applications.

4.1 Overview

As we observed in Chapter 2, purely history-based page scheduling methods are limited in the performance opportunities they can provide to applications running on hybrid memory systems. Instead, they must be augmented with more intelligent, predictive methods.

Why a solution with Machine Intelligence (MI)?

As we will further show in Section 4.2, the immediately observed memory access behavior is insufficient in capturing the necessary information that predicts future behavior for mak-

[1]The name is inspired by ancient Greek mythology, where Kleio is the muse of history, daughter of Mnemosyne, goddess of memory.

ing clever placement decisions. Yet, a larger window of accesses should allow the ability to capture the historic information (*long term access*), and also leverage the recent accesses (*short term access*) for effective page placement. However, understanding how to couple information from a window of past access history is not a trivial modeling process. For this reason we explore the use of advanced machine learning models, that provide *ready-to-use* mechanisms to handle temporal data capturing both short and long term page access patterns. Such techniques are reinforcement learning and deep neural networks (recurrent neural / long short term memory networks), which are currently widely explored to solve various systems problems, as we summarize in Chapter 8.

Overview of Contributions

The primary goal is to use machine intelligence to build a hybrid memory page scheduler that can bridge the performance gap between the current state-of-the-art history-based and an oracular page scheduler. We build a new page scheduler – **Kleio** – and we answer important questions concerning how to achieve an *effective* solution (i.e., one that maximizes the extent to which the performance gap is bridged), and a *practical* solution (i.e., one that can be realized while expending only a controlled or limited amount of resources on the typically compute-intensive machine intelligence processing tasks).

The specific contributions are the following:

- *Gap in current solutions*: We show the significant room for application performance improvement that is feasible in hybrid memory systems via clever data placement. This is due to the fact that predominantly used solutions, which look at recent memory access activity, are not computationally robust so as to capture complex page access patterns (Section 4.2).
- *MI-based page scheduling*: We identify Recurrent Neural Networks (RNNs) as an effective and practical technique for the page scheduling problem (Section 4.3). We

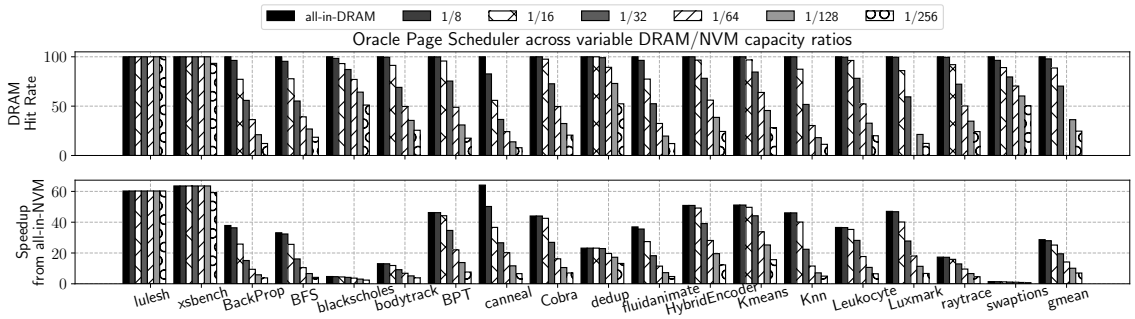
show that RNN training on a per application page granularity is highly accurate and leads to significant performance improvements even when applied to a subset of pages. While not exhaustively exploring all possible Deep Neural Network (DNN) algorithms, we present insights on the important trade-offs that must be considered when selecting an MI approach: its computational and space complexity and its applicability for the feature set which describes the page scheduling problem (Section 4.4).

- *Kleio*: We design **Kleio**, a practical, hybrid MI-based page scheduler. Kleio is hybrid because it combines existing history-based page scheduling, when such more lightweight methods are effective, with RNN-based machine intelligence, when history-based methods fail. Kleio is practical because it incorporates a new method for identifying pages where MI-based scheduling leads to most significant performance boost and prioritizing the use of system resources for these pages (Section 4.5).

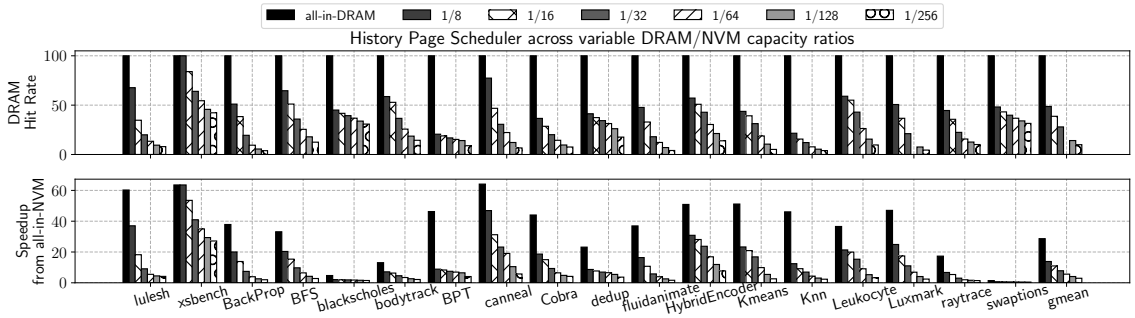
- *Performance improvements*: Using a range of workloads from popular suites, we show that Kleio can bridge on average 80% of the performance gap, that exists between the history-based page scheduling and oracular knowledge of the access pattern of a small set of cleverly selected application pages (Section 4.6).

4.2 Motivation

In this section, we present more detailed graphs that capture the performance of existing history-based versus oracular page schedulers for hybrid memories, as we summarized in Chapter 2. We show the performance that is achieved by such page schedulers across different capacity ratios of hybrid memory platforms. Recently emerged hardware platforms, such as Intel's Optane PMEM [1], are configured with 1/16 capacity ratio between DRAM and PMEM / NVM. We expect that future platforms, node-attached and rack-attached memories, will preserve substantial DRAM capacity, so as to offset the slower



(a) The Oracle page scheduler periodically migrates application pages such that DRAM hosts the pages with the highest access counts in the *current* scheduling epoch until capacity is full.



(b) The History page scheduler periodically migrates application pages such that DRAM hosts the pages with the highest access counts in the *previous* scheduling epoch until capacity is full.

Figure 4.1: Application performance for decreasing ratio of DRAM to NVM and fixed overall capacity to be the per application memory footprint.

access speeds of persistent memory, as well as reliability and write endurance and amplification issues [3]. For completeness we present performance numbers even over very limited DRAM capacities.

Figure 4.1a shows the performance achieved by an **Oracle page scheduler** across decreasing availability of DRAM capacity. Even in the case of a-priori knowledge of the workload's access pattern, the restricted DRAM capacity can severely impact performance, especially when it is available only in smaller amounts (e.g., 1/256 DRAM/NVM ratio). We also validate the observation [55] that the use of the minimum necessary DRAM capacity that is able to host the hot pages across the scheduling epochs (i.e., 1/8 in our case) can provide almost the same performance as if having infinite DRAM capacity (i.e., all-in-DRAM).

Figure 4.1b shows how the placement methodology of the current state-of-the-art **History page scheduler** can reduce performance up to **55%** (in the case of `lulesh`) and 13% on average. This is due to the fact that the history-based scheduler is built on the observation that applications preserve their page access pattern for certain time intervals, which may span across multiple scheduling epochs. Although this leads to good page placement decisions during such epochs, it fails to capture changes in the workload's memory access behavior. For example, there are times where the subset of hot pages may be completely disjoint between consecutive scheduling epochs, as the application transitioned into computation that involves data allocated in different memory areas. In this case, the performance impact is significant and makes a case for more intelligent data management using clever extrapolation of the past memory access pattern and not just the immediately observed behavior.

Takeaways

Current history-based page scheduling is not intelligent enough to boost application performance across capacity configurations of hybrid memory platforms. To address this, we choose to explore machine intelligence techniques given their ability to learn complex com-

binations of multi-featured information.

We aim to achieve two important **goals**:

1. Bridge the performance gap between the Oracle and History page schedulers.
2. Deliver low training and inference times by reducing the input problem space. This would allow the approach to be possibly integrated in an online solution.

In doing so, we contribute answers to the following **questions**:

1. Which machine intelligence technique to use (Section 4.3)?
2. How should we formalize the data input to the machine intelligence algorithm, so that it adheres to the purpose of predicting page access behavior to be used by a page scheduler (Section Section 4.4)?
3. How can we reduce the input problem space? Do all pages actually need machine intelligence-based management? How many are the pages whose timely placement in DRAM significantly boosts performance, while the History scheduler fails to properly manage them (Section 4.5)?

4.3 Choosing the Machine Learning Method

In this section, we explore the machine intelligence techniques that seem to be a good fit when designing an application page scheduler for data management over hybrid memory systems. The machine learning methods we consider are reinforcement learning and recurrent neural networks, that are widely used across similar systems problems, as we summarize in Chapter 8.

Reinforcement Learning

First, we explored deep reinforcement learning [56, 57, 58], a machine intelligence technique that enables an agent to learn through taking actions in a defined environment, in order to maximize a reward entity via the received feedback. In more detail, the page scheduler (*agent*) periodically interrupts the execution of the application to take an *action*, that is to migrate pages across the memory components. Then, the application resumes execution (*environment*) and during the next scheduling epoch (interrupt) the page scheduler receives its *reward*, that is the DRAM hit rate with the most recent page placement (*state*). In this way, the page scheduler learns the dynamic data layout that optimizes application performance across its runtime.

Why it is not a good fit. Although the approach of reinforcement learning seems to be a great fit into the problem description of a hybrid memory page scheduler, it cannot be practically used. This is due to the prohibitively large amount of possible actions the agent (page scheduler) can take. More specifically, a single action of a page scheduler involves taking a placement decision for each individual application page. For example, if there are two memory components and N pages, then there are 2^N possible placements, thus actions to choose from. Considering Table 4.1, that summarizes the number of pages across our pool of applications, N can be in the order of hundred thousands. In addition, if anything changes regarding the hybrid memory environment, such as the number of memory units, their capacity and relative access speeds, then different actions may be more beneficial and the reinforcement learning agent should be re-trained. Therefore, replacing the hybrid memory manager with a reinforcement learning agent makes the system solution dependent on the configuration of underlying memory hardware platform. In conclusion, the exponential action space and sensitivity to changes in the hybrid memory configuration, made us drop the approach of reinforcement learning for the context of our problem.

Recurrent Neural Networks

Another machine intelligence approach, which seemed appropriate for the purpose of the

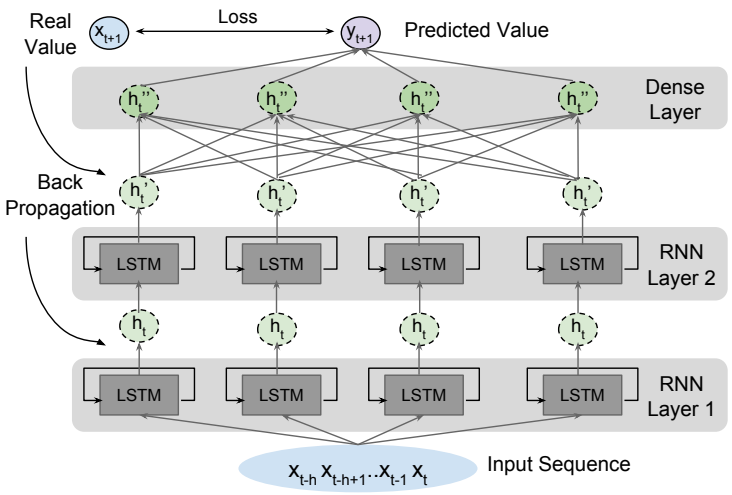


Figure 4.2: Example layout of a Recurrent Neural Network (RNN), using Long Short Term Memory (LSTM) neurons.

hybrid memory page scheduler, is Recurrent Neural Networks (RNNs). Different from reinforcement learning, where interaction with an environment facilitates learning, RNNs are able to find long-term dependencies in a sequence of data points and make predictions about future data behavior.

In the context of the page scheduler, these data points can be the sequence of pages accessed throughout an application execution time interval. The page scheduler can deploy an RNN in order to learn the page access pattern and make predictions about future page accesses. Using those predictions the page scheduler can determine which pages should be prioritized for allocation in the most appropriate memory component. For example, future highly accessed pages should be allocated in the lowest access latency memory technology. We choose to adapt this machine intelligence technique, since it has already been used to solve similar problems, such as hardware memory prefetching [59]. In contrast with reinforcement learning, where the problem space was growing exponentially to the number of application pages, in the case of RNNs it grows linearly with the number of pages. Furthermore, in Section 4.5 we show how it can be significantly reduced for the purpose of fast and efficient learning.

RNN Functionality. Next, we present the internal functionality of RNNs on a very high level. Currently, a widely used type of RNN is the Long Short Term Memory (LSTM) Network, that given a sequence of data points from time $t - h$ up to time t, can make a value prediction for time $t + 1$, where h is the length of retained history. For example, if the sequence represents the weather forecast of a city from April to November, the LSTM can make a weather prediction for December. In more detail, a single LSTM neuron takes the input sequence and converts it into an internal state h_t, via a non-linear combination of the weights and biases of its internal 'gates'. There are the 'input', 'output' and 'forget' gates that dictate what information gets filtered from the input and propagated towards the output. In this way, a single LSTM neuron is able to capture past data information into an internal state representation and make predictions about future data points.

An RNN can be constructed via the combination of multiple LSTM neurons on a single layer, stacked LSTM layers together with regular Dense layers, as depicted in Figure 4.2. The input sequence is split into subsequences of *history length* h, in a rolling window fashion. During a *training epoch*, all input subsequences are fed into the network, which then makes a single value prediction for each subsequence. The difference between the predicted and actual values is captured through the *loss function* and *back-propagated* into the network, where its weights and biases are getting updated according to the *learning rate*. Training can terminate when there is no reduction in the loss, thus the network cannot make any predictions closer to the actual value. In Section 4.5 we describe the network layout, hyper-parameter values and further fine-tuning techniques that will facilitate learning for the provided input data.

4.4 Choosing the Patterns to Learn

When using neural networks, an important step is choosing the features which describe the problem and are to be used as inputs. In this section, we discuss the representation of the data sequence related to memory access behaviors to be fed into the RNN and the

interpretation of the predicted value, as this is crucial for the training time and accuracy of the generated model. We further explore possible ways to reduce the input problem space and enable faster and more resource-efficient learning.

Input Data. The data we have available for each application is a memory access trace, as depicted in Figure 4.3. More specifically, it is the sequence of the page accesses that were serviced from main memory and not the processor's hardware caches, as they happened throughout the application run time. In Section 4.5 we describe in detail the way we acquire the trace and the exact information it contains.

Learning Objective. The aim of the RNN training is to be able to make predictions with respect to the number of future memory accesses, so as to aggregate the accesses on an application page granularity and then determine an ordering of heavily accessed pages. These predictions need to happen periodically, when the page scheduler is invoked, so that the appropriate page migrations are determined and executed. That is future hot pages need to be migrated to the memory technology component with lowest access latency.

Training Time. One of our main considerations is to enable fast learning via reduced training times and resource utilization optimized techniques. The duration of training models can be critical when considering use of machine intelligence in systems solutions, which to be practical, must operate within limited time and computational resource budgets. Undoubtedly the use of computationally robust technologies, like Graphics Processing Unit (GPU), Tensor Processing Unit (TPU), custom accelerators, can accelerate learning. However, our primary goal is to explore ways to enable faster learning via the training methodology, that can further be boosted via appropriate hardware.

Across Pages Prediction

The most intuitive way to learn from a memory access trace is to feed it 'as-is' into the RNN, following the x-axis in Figure 4.3. In this case, the RNN looks into a subsequence

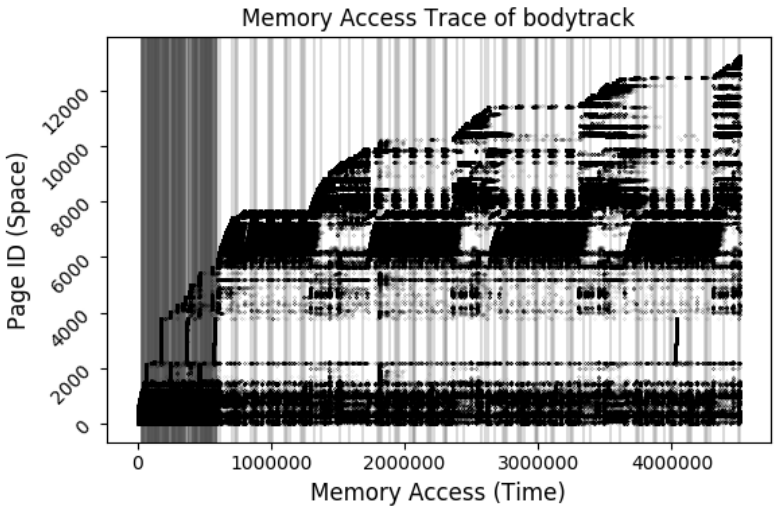


Figure 4.3: Example memory access trace, from the PARSEC suite. For every consecutive memory access (x-axis) we plot the page accessed (y-axis). The vertical gray lines correspond to the scheduling epoch time intervals.

of page accesses and predicts the page to be accessed next. Such an RNN use case is used by Hashemi *et al.* [59], for the purpose of prefetching future memory address accesses.

This approach has several *limitations:*

1. Large training time. To begin with, the input trace usually contains millions of memory accesses, especially at the data input scales of High Performance Computing (HPC) applications. This makes training time prohibitively large, in the order of couple days, at least when using the hardware setup described in Section 4.5.

2. Low prediction accuracy. Furthermore, when the output value space is significantly large (number of different pages), the RNN prediction accuracy tends to be low. Neural networks work better with normalized inputs (e.g., between 0 and 1 [59]). However, when normalizing hundred thousand values in such a way (total number of pages according to Table 4.1), there will be vast information loss. This is the reason why Hashemi *et al.* [59], choose to reduce the output value space (number of different memory addresses), by discretizing it into frequently appearing values (classes), and training different RNNs across clusters of the address space covered by the application. Most importantly, they accept top-

k predictions at a time, so as to increase the chances of a correct prediction. Although this is acceptable for the purpose of prefetching, it is not the case for a page placement decision, where a single prediction is needed, in order to accumulate the number of per page accesses.

3. Not an exact fit for the page scheduler description. The hybrid memory page scheduler operates periodically, aggregating the per page access counts during an application runtime interval referred to as scheduling epoch. Then the scheduler will determine the appropriate page ordering and issue the necessary migrations across the memory components. However, the number of memory accesses differs across the scheduling epochs, as it is visible by the vertical lines in Figure 4.3, where only 10% of the total memory accesses happened during the first half of the scheduling epochs. This is subject to the code executed during that time with respect to its computation to data access ratio and the technology parameters of the processor and memory regarding the time it takes to execute an operation, load data, etc. Throughout our application pool, we observe that just 10% of the total memory accesses happen, on average, throughout the first 37% of the scheduling epochs. Thus, there is no way to know before-hand how many accesses are going to happen in the next scheduling epoch, that is how far in the future the RNN should make predictions for (unless we train a different RNN for that purpose!).

In conclusion, we reject the idea to treat the input access trace as-is, given the restrictions described above. Next, we will see how we can extract the necessary information from the trace, so as to enable faster and accurate learning, that is also more suitable for the functionality of a page scheduler.

Per Page Prediction

Instead of predicting *which* page is going to be accessed next (across pages prediction), we flip the problem and explore the case of predicting *when* a page is going to be accessed next (per page prediction). So instead of predicting the y-value following the x-axis, we take

each y-value (page) and predict the sum of accesses across the scheduling epoch intervals on the x-axis. Thus, we propose training individual RNNs for every single application page. So, we feed into the per page RNN the sequence of access counts across the scheduling epochs and predict the number of accesses that the page will receive in the next scheduling epoch. In contrast with the prediction across page, the per page prediction:

1. Fits the page scheduler description. The above transformation of the input access trace fits exactly the functionality of the page scheduler, which will aggregate the page access counts on a scheduling epoch interval, so as to order frequently accessed pages and appropriately migrate them across the hybrid memory components.

2. Enables high prediction accuracy. Depending on the epoch duration and hotness of the page, the maximum number of accesses per epoch is in the order of hundreds, which is orders of magnitude less than problem space that the prediction across pages needed to capture, normalize and predict. Thus, this output value range is more suitable for RNN training.

3. Allows for low training times. Having a different RNN model per page, when the total number of pages can be in the order of hundred thousands, is similar to having a single RNN model that makes predictions across all these pages, as described earlier, since the input problem size remains the same, as depicted in Figure 4.3. Similarly to clustering techniques of the address space into memory regions and focusing on the frequently appearing memory addresses, as Hashemi *et al.* [59] did, there is scope to focus on the pages that are critical to application performance, which will significantly reduce the number of RNN models and overall training time, thus resource consumption.

4.5 System Design of Kleio

We propose **Kleio**, a page scheduler for hybrid memory systems, that leverages the existing state-of-the-art data management solutions and optimizes application performance by

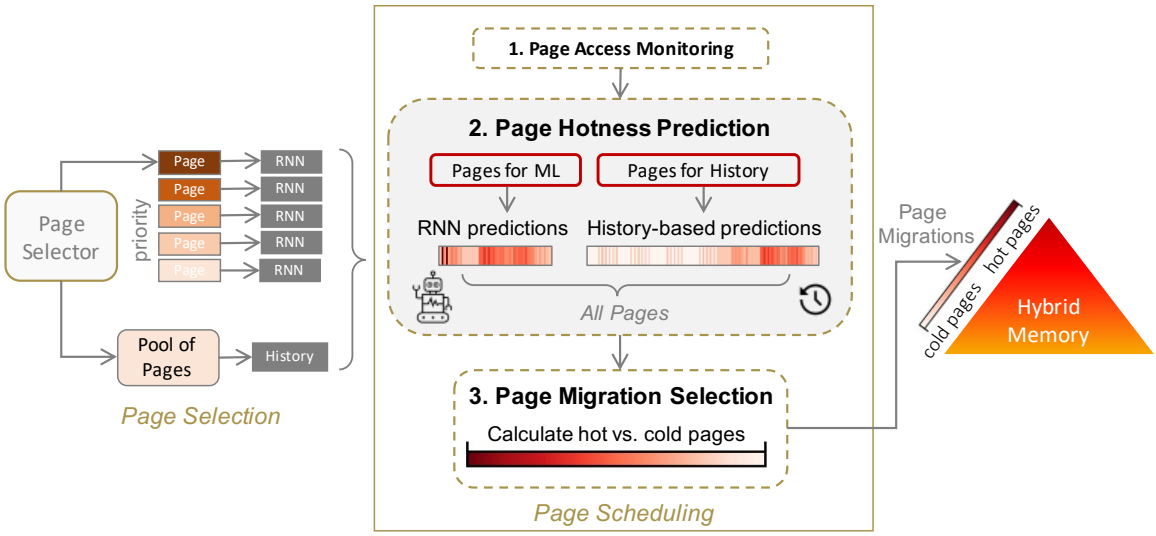


Figure 4.4: Kleio is a hybrid memory page scheduler, that combines the current state-of-the-art page placement methodology together with machine intelligence based management of the page subset, whose timely placement in the appropriate memory component is crucial for application performance.

delivering machine intelligence based placement decisions for a cleverly selected page subset. Figure 4.4, summarizes Kleio's internal functionality, which includes a page selection process and a page scheduling policy, described as follows:

During the page selection process Kleio:

1. Identifies a subset of the application pages that are important for performance, through its page selector component, described in detail later on.

2. Trains an individual Recurrent Neural Network (RNN) for as many of the important pages in their given order as it is allowed by the available resources for training. Kleio learns the patterns of per page hotness across periodic time intervals of the application runtime. Kleio does inference on the trained models to produce the associated RNN predictions, to be used during page scheduling.

3. For the rest of the pages, Kleio generates lightweight history-based predictions as what is used in existing solutions.

During the page scheduling Kleio:

1. Periodically monitors the page accesses and calculates the hotness of each page during the current scheduling epoch.

2. Uses a hybrid policy to predict the page hotness for the next scheduling epoch. Kleio uses the inferred RNN predictions for the pages selected for ML and history-based ones for the rest of the pages.

3. It then sorts the pages in descending hotness order, moving hot pages to DRAM and replacing cold ones, until DRAM capacity is full.

Page Selector Component

We next describe the page selector component of Kleio. Its design is driven by the following observations regarding the importance of correct page placement to application performance:

- There is only a certain subset of pages that needs more clever data management, than what the existing history-based solutions can provide. That subset is significantly small for limited DRAM capacity.

- Pages that need machine intelligence based management, can be ordered with respect to the performance impact of their placement into the appropriate memory component. We define a benefit metric that enables the page ordering, prioritizing pages with high access counts and number of misplacements by the History page scheduler.

- Intelligent management of the pages following the aforementioned ordering does not correspond to linear performance improvement. In contrast, intelligent placement for only (a small) part of them can bring most of the performance benefits we would get by applying intelligent placement across all application pages.

We define a *misplacement* of a page by the History scheduler, when at the start of a scheduling epoch, a page was supposed to be allocated in DRAM, but it was not, because of wrong

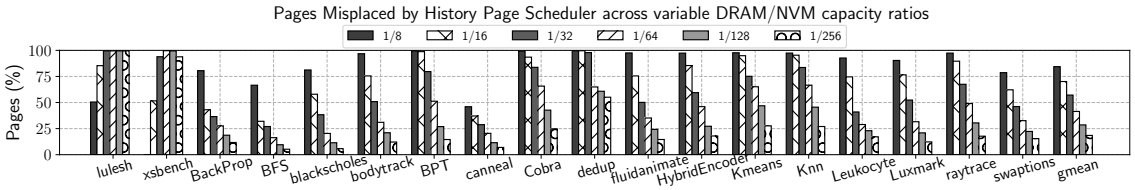


Figure 4.5: Percentage of pages misplaced at least one time across the scheduling epochs by the History page scheduler. This is the set of pages that need machine intelligence based management. This observation is crucial since it highlights that the problem space of per page RNN training can be significantly reduced as the size of available DRAM does.

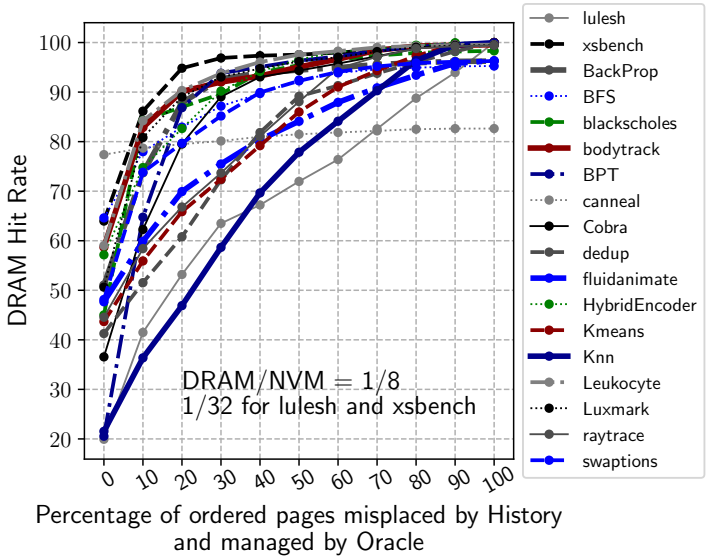


Figure 4.6: DRAM hit rate when an Oracle Page scheduler manages the misplaced-by-History pages and the History page scheduler manages the rest. Pages are ordered in descending performance benefit. Clever management of even a small percentage of these pages, can give most of the performance benefits we would have by managing cleverly all pages.

hotness prediction. Figure 4.5 depicts the percentage of application pages, which are misplaced by the History page scheduler, at least during one scheduling epoch, across reducing DRAM capacity. This signifies the set of pages that need more clever management. In combination with the actual per application page count summarized in Table 4.1 and the limited DRAM capacity, the number of such pages can be in the order of hundreds. This drastically reduces the problem space of RNN training.

However, even by reducing the number of such pages, there still may not be enough resources or time to train per page RNN models. Thus, there needs to be a priority ordering of these pages, so as to cleverly manage those that can give the biggest application performance boost, when timely placed into DRAM. For this reason, we capture the importance of a page in the *benefit* that its correct placement would provide to application performance. The benefit increases with the hotness of a page, similarly to prioritizing frequently accessed pages for DRAM allocations across the scheduling epochs. However, we also need to take into account the number of misplacements-by-the-history-scheduler each page received, as the timely placement of a page in DRAM together with its hotness, will boost performance. To this extent, we define the following benefit factor for prioritizing the pages in need of RNN training.

$$\texttt{Benefit} = \texttt{Number of accesses} \times \texttt{Number of misplacements}$$

Next, we capture the range of application performance boost we would get, if we could manage part of the aforementioned misplaced pages with the Oracle page scheduler and the rest with the History page scheduler, since it already places hot pages in DRAM in time. This will set the upper limit of the performance boost we can get with RNN training of the misplaced pages. Figure 4.6 captures this performance improvement. When the Oracle scheduler manages 0% of the misplaced pages, it is equivalent to all pages being managed by the History scheduler, thus is the lowest bound of performance. In contrast, when the Oracle scheduler manages 100% of the misplaced pages, it is equivalent to the Oracle managing all the application pages, since the rest of the pages were not misplaced

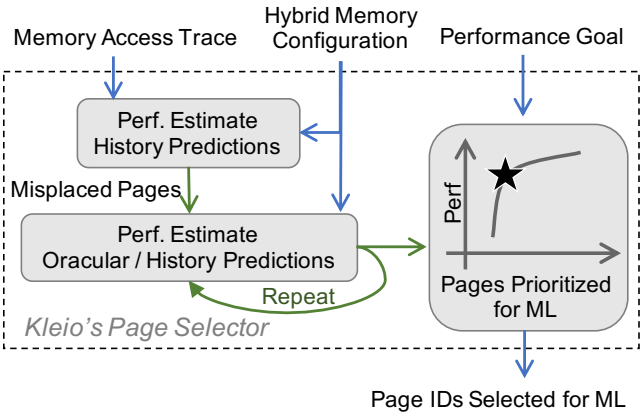


Figure 4.7: Kleio's page selector component is able to identify the pages whose machine intelligence based management will bring the highest application performance improvements, while enabling focused and practical RNN training.

by the History scheduler. That sets the upper bound of performance we can have on a per application basis. We observe a non-linear relation between the set of pages and the performance enhancement. This is due to the page ordering with respect to the defined benefit factor, that is able to prioritize hot pages, whose timely DRAM allocation guarantees significant performance improvement. For example, in the case where the curve shows a distinct knee, the pages after the knee, received far less accesses, thus their timely DRAM placement will bring trivial benefit to the DRAM hit rate.

Page Selection Pipeline. Kleio's page selector component captures the above observations and provides insight into the relation between the number of pages that need RNN training together with a best case scenario of corresponding application performance improvement. Figure 4.7 summarizes the work flow of the page selector component, its internal functionality, input parameters and generated output. Kleio utilizes models to estimate application performance depending on the configuration of the hybrid memory (i.e., number of components, capacity ratios, access speeds) by simulating the page scheduling process. First, Kleio's Page Selector generates an application runtime estimate when using history-based predictions during page scheduling. In this way, Kleio captures the number of times each page is misplaced (e.g., moved to slower memory even though the page was hot), due to

mispredicting the page hotness when using purely historical information. Kleio couples this information together with the page hotness into a benefit factor, that determines the page priority for machine learning-based management. Then, following this page priority order, Kleio runs the estimate model repeatedly to generate a performance curve when simulating perfectly accurate (oracular) predictions for pages with high priority and history-based ones for the remainder of the pages. Given a performance goal (e.g., 90% of all possible performance improvements), Kleio's Page Selector outputs the identifiers of the corresponding pages in descending priority order. After this page selection process, Kleio will train per page RNNs for the selected pages.

System Implementation Details

In the remainder of this Section, we describe further details about the implementation and configuration of the Recurrrent Neural Networks used as part of Kleio's system design.

Neural Network Layout. Figure 4.2 gives a visual representation of the RNN we deployed, consisting of LSTM neurons. The network consists of two stacked RNN layers with *128* LSTM neurons each, followed by a Dense Layer. The history length is *16*, thus the input data series is split in sequences of length 16, on a rolling window fashion, while *70%* of them are used as a training dataset and *30%* of them for validation. The neural network tries to minimize the *mean squared error* (loss) between the predicted and actual values, using the Adam [60] optimizer on a learning rate of *0.001*. The model training stops, if the loss for the validation dataset is not reduced for *20* consecutive training epochs. The duration and accuracy of the trained models is reported later in this Section.

Neural Network Data Manipulation. As described earlier, the RNN input corresponds to a sequence of per page access counts during consecutive scheduling epochs, while the output is the predicted number of accesses the page will receive during the next epoch. The predicted number will then be used by the page scheduler to determine the hotness order across all pages. Thus, there is room for the prediction to be slightly different than the

actual number of accesses, as long as it will not influence the hotness order of the page, and therefore its placement decision, on the particular scheduling epoch.

Therefore, we normalize the input sequences between 0 and 1, since RNNs work better in this case as observed by Hashemi *et al.* [59] and then denormalize the data for the final prediction. Different from [59], there is no need for us to make predictions over distinct integers, treating the prediction problem as classification. Our experiments with the classification approach, highlighted the possibility of misprediction with a great margin from the actual value and gave reasoning as to why Hashemi *et al.* [59] chose to consider top-k predictions at a time. Although this approach works great with the prefetching logic, where more data can be prefetched even if they do not end up being accessed, this is not necessary for the purpose of our predictions.

It is important to observe that, even though the input data (memory access trace) is the same between this work and [59], the prediction use case transforms the way they should be manipulated for RNN training and the accepted level of prediction accuracy.

Code Implementation. We use the Keras [61] high level API to deploy the described RNN layout, using the existing implementations for the LSTM neurons, the network layers connectivity, the Adam optimizer and model training, applying any default hyper-parameter values if not explicitly mentioned above. The backend RNN execution engine is Tensorflow [62].

4.6 Evaluation

Experimental Methodology

We first describe the specific methodology used to evaluate Kleio.

Applications. Table 4.1 summarizes the set of workloads we used to motivate and evaluate Kleio, spanning across domains with representative computation kernels and stressing different components of the system (e.g., memory, CPU, GPU). We included workloads

Table 4.1: Workloads used for evaluation. Number of pages $\times$ 4 KiloBytes will be the total application memory footprint. Scheduling epochs is the number of times that the page scheduler was periodically invoked within the application runtime, so as to reposition pages across the hybrid memory subsystem.

Application	**Suite**	**Domain**	**Pages (4 KB)**	**Sched. Periods**
Lulesh	CORAL	Hydrodynamics	847,252	206
XSBench	CORAL	Monte Carlo	136,098	856
blackscholes	PARSEC	Finance	8,033	302
bodytrack	PARSEC	Comp. Vision	13,259	389
canneal	PARSEC	Engineering	56,974	398
dedup	PARSEC	Storage	131,259	657
fluidanimate	PARSEC	Animation	54,286	333
raytrace	PARSEC	Visualization	22,890	347
swaptions	PARSEC	Finance	12,633	491
BackProp	Rodinia	Pattern	35,083	117
BFS	Rodinia	Graph	27,396	26
BPT	Rodinia	Filesystems	142,923	485
Kmeans	Rodinia	Data Mining	70,783	87
Knn	Rodinia	Data Classifier	84,691	118
Leukocyte	Rodinia	Medical	56,580	180
Cobra	Windows	Video Transcode	83,720	168
HybridEncoder	Windows	Video Transcode	73,787	178
Luxmark	Windows	Image Creation	53,491	108

from the CORAL [63] suite, the PARSEC [64] suite utilizing the simlarge input sizes, and Rodinia [65], with the default input data sizes. Finally, we also included few Windows desktop applications. Concerning the memory footprint of these applications, Table 4.1 includes this information as a multiple of 4 KB pages, that gives a range of couple hundred MBs. These memory footprint sizes, though small in the context of real systems, are significant relative to the cycle-level memory simulation environment, and adequately capture the use case where the data will span across multiple main memory components, due to the limited capacity of available DRAM in future hybrid memory systems. Regarding the application runtime, it is again summarized in Table 4.1, as a multiple of the scheduling epoch intervals, when the page scheduler is periodically triggered throughout application execution. Our applications serve a variety of short and long running executions. Due to the difference in the trace collection methodology, for the CORAL workloads the scheduling epoch interval is 1 second, whereas for the rest is 0.01 seconds.

Memory Access Trace Collection. For each application we collect detailed traces of the data accesses that missed the last level of processor hardware caches and resulted in main memory accesses. For the CORAL workloads we used the Instruction Based Sampling (IBS) that is available on AMD's processors. This mechanism samples every Nth micro-operation, that goes through the processor's pipeline, out of which we filter the loads and stores. For the rest of the workloads, we collected unsampled traces for memory accesses that miss the last level cache on a system with an AMD A10-5800K Accelerated Processing Unit (APU) clocked at 3.8GHz and 16GB memory. The information included for each individual access is a timestamp, the physical and virtual memory address, the CPU core ID, the application thread ID, whether the access was a load or a store and a hit or miss. For the purpose of our analysis, we extract the 4 KB virtual page ID, that corresponds to the virtual memory address accessed and we group memory accesses into scheduling epoch intervals according to the timestamp, as depicted in Figure 4.3.

Hybrid Memory System Simulation. We simulate a hybrid memory system that contains a

Table 4.2: Technology parameters used in the simulated hybrid memory system, differentiating for Reads (R) and Writes (W) and sequential versus random accesses.

Technology	**R/W BW (GB/s)**	**Seq. & Rand. R/W Latency (ns)**
DRAM	19.2/19.2	8/8 & 50/50
NVM	10.24/1.024	8/8 & 100/1000

fast memory component (i.e., DRAM) and one with lower access latency (i.e., NVM). Both memory technologies serve as flat main memory, as they are part of a continuous physical memory address space. Table 4.2 summarizes the technology parameters of the simulated memory types. The capacity of the memory system is assumed to be the application's memory footprint. For example, when we refer to a DRAM/NVM capacity ratio of 1/16, we mean that DRAM will have space to accommodate 1/16 of the application pages and NVM will service the rest.

Apart from gathering the DRAM hit rate as an application performance metric, we also use the analytical model used by Meswani *et al.* [13] to extrapolate the application runtime, based on the number of accesses that are serviced from DRAM and NVM appropriately. In the case of the CORAL workloads, the number of accesses is properly adjusted based on the sampling rate. The model uses the Leading Loads method, which splits the application runtime into the time to perform computations and the time to satisfy memory requests, via the use of hardware performance counters. Regarding the time to service a memory request, the method maps it to the time spent servicing the leading (first out of many) load request that misses the last level hardware cache. This load time depends on the memory technology that serviced the request (e.g., DRAM versus NVM), whose differences are summarized in Table 4.2. This gives us a worst case performance estimate, since it does not take into account actions that reduce latency, such as parallel computation or prefetching. Also, we assume dedicated DMA engines that allow seamless page migration, which is overlapped with the computation, as explored in [66, 67].

Hardware testbed for training ML models. We conduct experiments using an AMD machine with 512 GB memory and 64 Opteron™ 6370P CPU cores of 2 GHz each. CPUs

have been used to accelerate RNN-based deep learning models [**216077**]. Kleio speeds up the training by intelligently selecting to train the application pages that will bring actual performance benefits. Instead, a more naive approach would rely on accelerators and rack-scale size machines in order to accommodate RNNs for all pages, wasting resources for training models whose predictions have trivial performance impact or can be achieved by simple history-based policies.

Evaluation of Application Performance

First, we evaluate the accuracy of Kleio's RNN training with respect to the corresponding application performance improvements, which is what Kleio promises to deliver. As a reminder, Kleio identifies the pages that are misplaced by the History page scheduler and applies RNN training in order to get predictions of their per epoch access counts and determine the global page hotness order for prioritizing DRAM allocations. If the RNN predictions are extremely accurate, then it would be equivalent to having an Oracle page scheduler manage the misplaced pages. To this extent, Figure 4.8a depicts the performance that Kleio can achieve when applying RNN training to **100 pages** in the order defined by its page selector component, for a given DRAM/NVM capacity ratio. We fix DRAM/NVM=1/32 for the CORAL workloads and DRAM/NVM=1/8 for the rest, which is the capacity ratio for which the clever management of even a small number of pages, can bring significant performance improvements (Figure 4.6). Performance is normalized between 0%, when all pages are managed by History page scheduler and 100%, when the selected pages are managed by Oracle and the rest by History. In this way, we can understand the degree to which the RNN predictions are sufficiently accurate, so as to provide all the possible performance improvement.

We observe that in most cases, the RNN predictions are sufficiently accurate to bring **80%** of the possible performance improvement, on average and more than **95%** for half of the applications that we considered. Unfortunately, there are cases such as `bodytrack` and `raytrace`, where less than 50% of the possible speedup is achieved, in which case

more pages need to be trained so as to further provide significant speedup.

Overall, we prove that the accuracy of the RNN predictions is such that it can deliver application performance similar to what would be possible with oracular knowledge of the access frequency. Kleio's page selector is useful, so as to determine the number of pages that is necessary to train in order to observe significant performance improvements.

Evaluation of Prediction Accuracy

We next present the actual prediction accuracy of the per page RNN training. Figure 4.8b depicts the distribution of the Mean Absolute Error (MAE), in boxplot representation, between the cumulative per epoch page access counts and the actual values, across the trained application pages. For example, mean MAE of 30, means that the RNN predicted 30 more accesses on average per epoch per page. On the same graph, we treat the decisions of the History page scheduler also as predictions and plot the corresponding MAE. The History page scheduler predicts that on the next scheduling epoch a page will receive the same access counts as to those of the current epoch.

As expected, the History prediction can be far from reality, as it is common for a page to convert from being frequently accessed to not being accessed at all on two consecutive epochs, thus the prediction MAE can be significantly high. In contrast, the RNN is able to make better predictions via the efficient LSTM learning, although still they may seem not as accurate enough. However, even if the per epoch access count prediction is not extremely accurate, as long as it does not affect the correct global page hotness order and actual page placement, there will be no application performance impact of the prediction. This is highlighted in Figure 4.8a, where for example `Luxmark` has a mean MAE of 50, though still achieves 85% of the possible performance improvements.

Figure 4.8c, further strengthens the above statement by showing the percentage reduction of page misplacements achieved by Kleio for the selected trained pages, compared to the History page scheduler across all pages. Although, Kleio still misplaces the selected pages on some scheduling epochs, the per page access count during those epochs is not

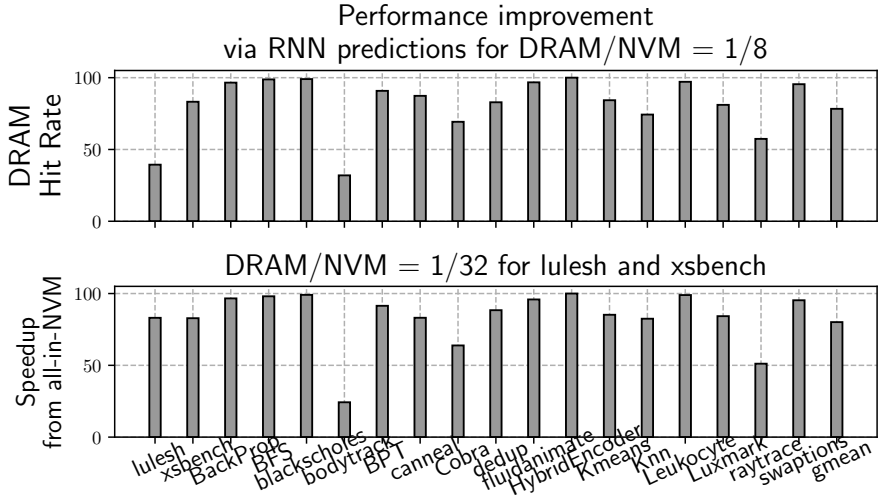
big enough to drastically impact the DRAM hit rate. Thus, Kleio manages to reduce on average 85% of the selected pages misplacements across the application lifetime.

Resource Utilization

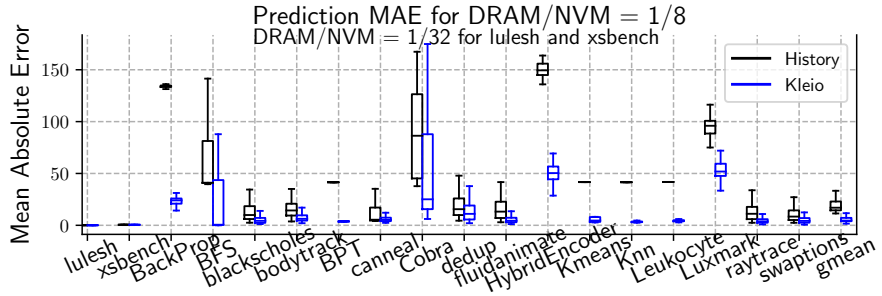
RNN training goes on until there is no further reduction of the loss over the validation data for a certain number of training epochs. The duration of the training is primarily affected by the network layout itself, that is the hyperparameter values and the length together with the number of the input sequences. Thus, the more training data the longer it takes to learn. Since we perform training on a per application and per page granularity, the number of input sequences is the number of scheduling epochs, divided by the history length hyper-parameter. Looking back at Table 4.1, this number will be in the order of couple hundreds, which enables fast training times.

More specifically, we report the following average metrics across pages and across applications, for the given hardware testbed described earlier. Training lasts on average for **120 training epochs**, that translates into a time duration of **2 hours** per model, when all models are trained at the same time, utilizing all system's resources. As far as memory utilization during training is concerned, the maximum observed per model was in the order of **tens of GBs**. Finally, regarding the storage overheads of saving the models after training, for the purpose of future inference and analysis, using the Hierarchical Data Format (.hdf5) available from the Keras library, it was less than **0.5 MB** per model. Regarding the resource utilization for the purpose of inference, it was trivial and the duration instantaneous (3-4 seconds).

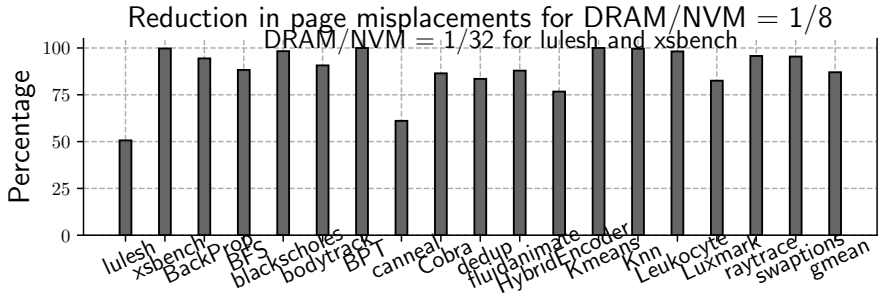
Putting all this information together, there is no doubt that the hardware resource requirements of RNN training are significant, especially as far as memory consumption is concerned. However, training times in the order of couple of hours are generally considered to be low, for machine intelligence purposes. As we summarize in Chapter 8, there is a plethora of software and hardware solutions that promise to accelerate ML training times. Regarding the use of CPUs for training, in particular, DeepCPU [68] is a library-level so-



(a) Performance achieved by machine intelligence based management of the first 100 most important to performance pages, while a History page scheduler manages the rest. Hit rate is normalized between 0, that is the worst-case where all application pages are managed by History and 100, that is the ideal case where Oracle manages these 100 pages. Kleio can deliver over 80% (gmean) of the performance improvements that an Oracle page scheduler would for the selected pages.



(b) Prediction accuracy of the number of access counts across the scheduling epochs for the selected trained pages.



(c) Reduction in the number of page misplacements via the achieved RNN prediction accuracy, compared to the History page scheduler.

Figure 4.8: Evaluation of the application performance Kleio can deliver via the achieved levels of prediction accuracy and reduction in page misplacements across hybrid memory.

lution that improves the RNN performance on CPUs by an order of magnitude. Using such solutions can drastically reduce the learning overheads of Kleio. Either way, the user may be limited with respect to how many per page models can train, given the available system resources.

Kleio has provisioned for the case of limited hardware resources through its page selector component, that provides the user with information regarding which pages to prioritize for RNN training and the corresponding expected application performance improvements.

Reaching our initial Goals.

1. Kleio promises to bridge the performance gap between the Oracle and History page schedulers, delivering on average 80% of the theoretically possible performance when managing selected pages, through the achieved RNN prediction accuracy.

2. Kleio delivers low training and inference times, via deploying RNN models for cleverly selected application pages, whose timely placement in DRAM significantly boosts performance. Kleio shows that not all pages are in need of intelligent data management, drastically reducing the input problem space.

4.7 Chapter Summary

In this chapter, we describe Kleio, a page scheduler with machine intelligence for applications that execute over hybrid memory systems. In an effort to deliver a practical solution, Kleio reveals that an approach that replaces resource management with a machine intelligent component, like a reinforcement learning agent, will not be scalable and robust to hardware configuration changes. Given the massive memory footprints of applications executing over hybrid memories, Kleio identifies a small page subset, whose machine intelligent management boosts application performance. Then, Kleio deploys Recurrent Neural Networks to learn page-level access patterns, while using lightweight existing history-based predictions for majority of the application pages. In this way, Kleio bridges on av-

erage 80% of the relative performance gap between existing and oracular solutions. While Kleio shows great promise that a machine learning-based approach can be highly effective and practical, there are concerns that remain regarding the non trivial operational and learning overheads associated with such a solution. The remainder of the thesis explores ways to minimize such overheads and missed opportunities to further boost application performance, that are complementary to the use of machine learning.

CHAPTER 5
FINE-TUNING CRITICAL MANAGEMENT OPERATIONS WITH REUSE INSIGHTS

So far this thesis contributes the system design choices that enable practical foundations for the integration of machine learning in hybrid memory management, bridging the performance gap left by current history-based methods. In this chapter we explore ways to further boost application performance and system resource efficiency by identifying a missed opportunity in maximizing the benefits from hybrid memory due to empirical configurations of the system's operational frequency. To this end, we propose **Cori**[1] [32], a system-level tuning solution for hybrid memory management solutions that this thesis improves upon. The chapter reveals insights regarding the relationship among the operational frequency and the application data reuse that are then used to build a lightweight and highly effective tuning tool. The described improvements unlock new performance levels, that can be further boosted with the use of machine learning-based management that this thesis contributes.

5.1 Overview

Regarding current hybrid memory management solutions, while a significant body of research focuses on optimizing the selection of *which* data to move, there is little insight towards *when* that data should be moved. Focusing on the latter, Table 5.1 summarizes the operational frequencies of related data tiering solutions, whose difference in time ranges four orders of magnitude. These values are *empirically* tuned to meet the performance requirements of the specific pool of applications evaluated for their respective systems.

[1]The name is inspired by the ancient Greek mythology, where Cori (short for Terpsichore) was the muse of dance and daughter of Mnemosyne, the goddess of memory.

Table 5.1: Frequency of data monitoring and movement across existing solutions mapped to our simulation-based analogy.

Solution	Period Duration	Requests per Period
Thermostat [12]	10 sec	100,000
Nimble [14]	5 sec	50,000
Ingens [28]	2 sec	20,000
HMA [13]	1 sec	10,000
Hetero-OS [15], -Visor [20]	0.1 sec	1,000
Kleio [31]	0.01 sec	100
Unimem [9]	MPI phase	N/A

Empirical tuning of page scheduling frequency can miss significant performance improvements by not testing certain frequency ranges in an effort to minimize tuning overhead. For example, a common approach [13, 18] is to experiment with period durations that are an order of magnitude apart, e.g., 0.01 sec, 0.1 sec and 1 sec, so as to identify in only three trials which offers the highest DRAM hitrate while maintaining reasonable data movement overhead. On the other hand, exploring all frequency choices leads to impractical tuning overheads. In addition, the periodic solutions in Table 5.1 fix their operational frequency at the system-level, so that they do not have to repeat the empirical tuning for every application. However, this can potentially leave a significant amount of unexploited performance for applications with data access behaviors and sizes that the empirical tuning did not consider. Another approach is to completely rely on the application to explicitly control data allocation and movement, via use of specialized pragmas or `malloc`-like APIs. Such modified applications then explicitly control how the underlying system-level solution maintains the necessary state to dynamically manage data tiering across hybrid memory [6, 8, 9, 10].

Problem Statement. Impractical tuning overheads and lack of insight force existing data tiering solutions to rely on empirical tuning of their operational frequency, or on application-level modifications suitable for specific execution models and APIs. As a result, for general scenarios where modifying the applications is not appropriate, there can be significant levels of performance that existing data tiering solutions do not realize across applications,

due to their empirically-tuned and fixed operational frequency.

Contributions. To address this, we propose **Cori** – a system-level solution for tuning the operational periods in page schedulers, that maximizes the effectiveness of the schedulers in terms of application performance and platform efficiency, and achieves that with low tuning overheads. Cori operates in an application and runtime-agnostic manner, and relies on observation-based insights to guide the frequency tuning process to a small number of viable candidates. We demonstrate that Cori is effective, *irrespective* of the data access behavior and page scheduling effectiveness, and can be practically integrated into the existing hybrid memory management software stack.

The specific contributions are the following:

- We demonstrate that current data tiering solutions can experience 10%-100% performance loss due to sub-optimal choice of their operational frequencies (Section 5.2).
- We identify a relationship among observable application properties – their data reuse – and the favorable scheduling periods (Section 5.3).
- We describe the design of **Cori** and its frequency tuning methodology, for a simulation-based prototype and in real system settings. (Section 5.4). The implemented code base is open sourced[2].
- We evaluate Cori, demonstrating its ability to identify operational frequencies which realize performance improvements within only 3% from the ideal frequency selection, on average, across applications and page scheduling variations. Cori achieves this with 5× fewer number of tuning trials, compared to insight-less tuning approaches (Section 5.5).
- We validate Cori's insights, effectiveness and practicality on a real hardware testbed with DRAM and Intel's Optane DC PMEM (Section 5.5).

[2]https://github.com/GTkernel/cori-sim.git

5.2 Motivation

Performance Gap

Comparison with existing solutions aims to capture the application performance impact caused only by the selection of *when* to move data, not which and how much data to move. For this purpose we assume the page scheduling policies described in Chapter 3 and evaluate upon the data movement frequencies of existing solutions, as summarized in Table 5.1. Since these proposed values vary across orders of magnitude, we create corresponding period durations, summarized in Table 5.1 that map to the hybrid memory simulation environment described in Chapter 3.

Next, we capture the application performance gap created by using these proposed frequencies as opposed to an optimal frequency across a wide range of data access patterns. Figure 5.1 captures application runtime slowdown from the case of an optimal frequency that provides best performance, together with the corresponding amount of data moved as a percentage of the application's memory footprint. The performance of our proposed solution Cori is also included in the figure, but will be further analyzed in Section 5.5.

The proposed frequencies create a 10%-100% performance slowdown compared to the performance achievable with a best-case frequency, on average, across applications and page schedulers. This makes a case for the need for a more robust tuning approach than the empirical one. Taking a closer look, we observe that no single frequency works best across applications and page schedulers. In more detail, predictive vs. reactive page schedulers experience the lowest slowdown, on average, for frequencies that are an order of magnitude apart, that is a period duration of 1 second proposed by `HMA` vs. 10 seconds by `thermostat`, respectively. Additionally, the frequency that works best on average for a certain page scheduler may not provide best performance across *all* applications. For example, the lowest slowdown for a reactive page scheduler provided by `thermostat` is not the best choice for `pennant`, `lud`, `hotspot` and `kmeans`. In particular, it incurs

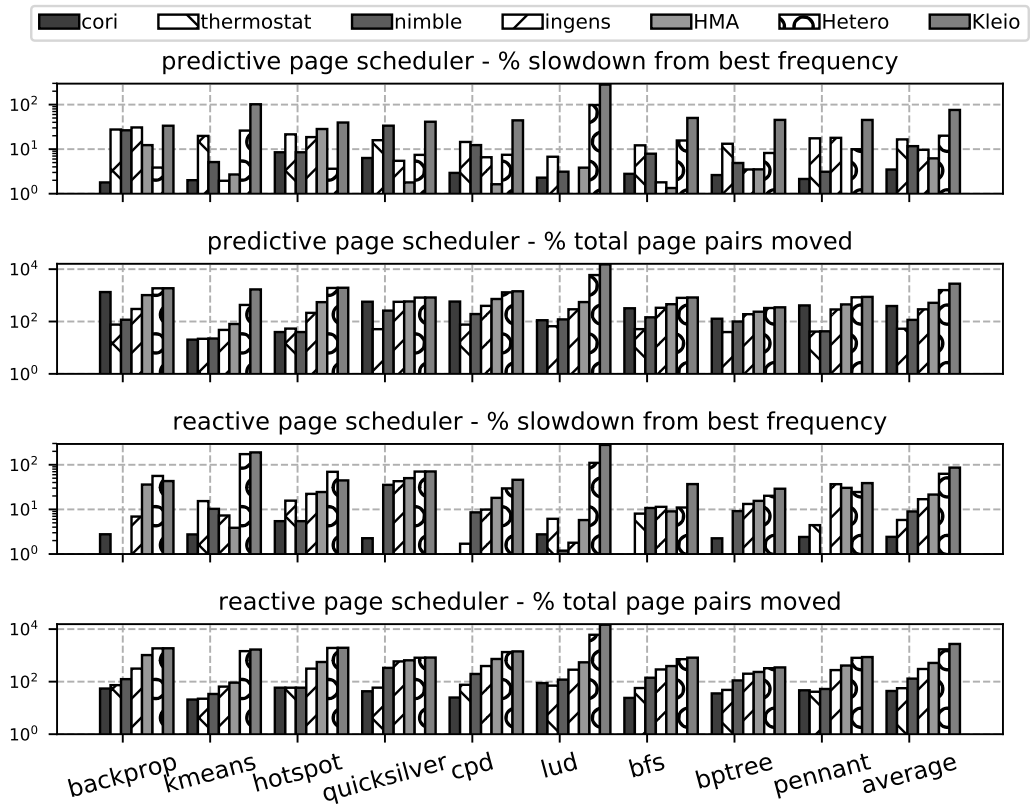


Figure 5.1: Performance comparison of a predictive and reactive page scheduler across operational frequencies of existing solutions and the proposed solution Cori, given a simulated hybrid memory system with DRAM and PMEM at a 20%:80% capacity ratio.

an average 8% slowdown from the respective best *proposed* frequency, that is additional to the slowdown from the best frequency itself.

Takeaways. This initial experiment validates our initial observations [69] and reveals that frequencies proposed by existing solutions leave a significant performance gap of 10%-100% across applications and page scheduler designs. No single proposed value works best across all applications and page schedulers. Therefore, there is an opportunity to close this performance gap with a more insightful tuning approach.

Tuning Overheads

Existing solutions choose to empirically tune their page scheduling frequency and fix it

across applications, to avoid the non-trivial tuning overheads of fine-grained frequency exploration. Stated more formally, an empirical tuning approach has $O(1)$ time complexity, since it chooses upon a constant set of frequencies. The choice of the frequencies themselves is critical, since an insight-less selection can lead to the aforementioned performance gap.

An exhaustive tuning approach has $O(N)$ time complexity, because the number of possible frequencies grows linearly with the application runtime. For example, the possible period durations for an application that generates N memory requests in total, are the windows of any length between $[1, \frac{N}{2}]$, assuming that a page scheduler should run for at least two periods of $\frac{N}{2}$ requests each. Similarly, if we consider the time domain instead of the memory request domain, the number of possible period durations is such that is splits the application runtime at multiples of a timestep, where a timestamp could be related to the Linux scheduling time slice, for instance.

The need for some insight. The long runtime of applications that require massive hybrid memory systems makes an exhaustive tuning approach completely impractical. By using a more insightful tuning method we can drastically reduce these overheads, and also eliminate the performance gap caused by a poor choice of migration frequency made by empirical selection approaches.

5.3 Data Reuse Insights

We perform the aforementioned exhaustive tuning approach to extract insights. We select applications with a wide range of data access behavior. Figure 5.2 shows a visual representation of their memory access patterns, as analyzed by the collected traces. We observe the strided array traversals of `backprop` and `quicksilver` vs. the distinctly shaped sparse tensor traversals of `cpd`, the triangular multiplication in `lud`, and the irregular memory accesses of `pennant` over a fixed number of repetitive cycles.

Page Reuse Distance. The top graphs in Figure 5.3 depict information on data reuse. In the

context of these analyses, we use page reuse distance as a measure for page reuse, where the page reuse distance is the number of memory accesses that are issued to other pages, between two consecutive accesses to a particular page. There is a clear connection between the page reuse distances and the access patterns in Figure 5.2. For example, for `backprop` the reuse distance of 20,000 requests maps to the gap between the large access strides, and it appears 15 times since there are 16 strides. In contrast, the decreasing appearances of page reuse distances for `lud` and `pennant` correspond to the triangular array traversal and random access behavior, respectively.

Relation of Performance and Data Reuse. The bottom graphs in Figure 5.3 capture the application runtime slowdown from the case of infinite DRAM capacity and from the case of optimal frequency selection, across all possible period durations for predictive and reactive page schedulers. The x-axis is aligned with the histogram (top graph) and aims to capture the relation between the page reuse distances and page scheduling period durations.

We observe that predictive page schedulers, which make a better selection of which pages to move, provide best application performance for much shorter periods than reactive ones. However, irrespective of the page scheduler's effectiveness, very short periods create a significant aggregate data monitoring and movement overhead, as also shown in Figure 5.1. In addition, arbitrarily long periods do not allow the page scheduler to react promptly to changes in the access pattern behavior, thus create insufficient data movement to dynamically improve the data tiering. Moreover, the effectiveness of reactive page schedulers suffers at periods whose length is shorter than the page reuse distances with significant appearances, incurring an average of 50% additional performance slowdown compared to predictive schedulers. For example, this is the case for `backprop` when periods are shorter than 20,000 requests per period, which is the page reuse distance of its strided access pattern. The scheduler's effectiveness drops because its reactive design identifies as hot pages the ones that correspond to a certain part of the access stride, then moves them to the limited DRAM capacity, but they will not be accessed in the next period, when the rest

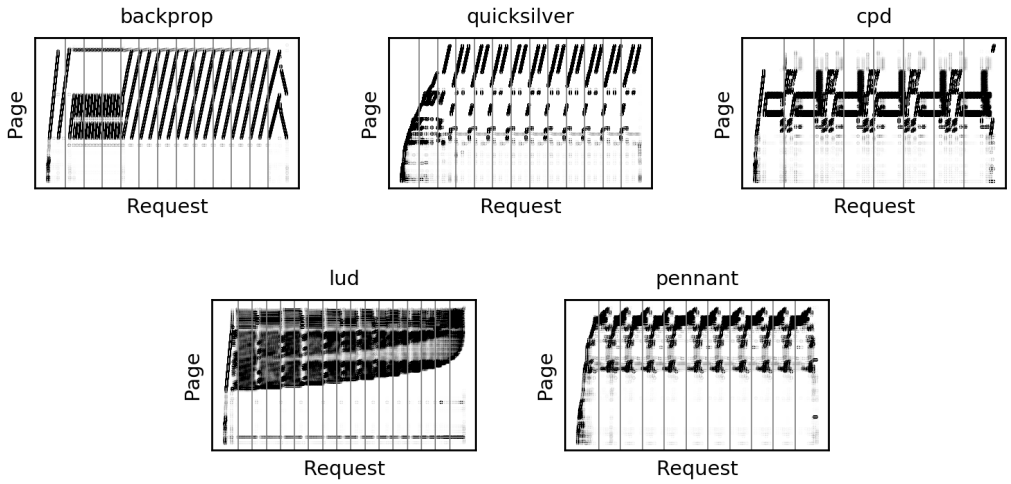


Figure 5.2: Representative memory access traces. The vertical lines correspond to the fixed period boundaries that provide best performance, as selected by Cori.

of the pages of the stride will be accessed. Such reactive page scheduling approaches are more effective when they operate over larger windows of access history, enabled either by longer periods or longer history of shorter periods. Regardless, the time window of access history should be large enough to not 'break' the data reuse.

Lessons learned. This extensive application performance characterization shows a clear relationship among the data reuse times and the page scheduling period durations which provide best performance. Reactive page schedulers benefit from periods that *don't break the data reuse*, to make better page migration decisions. Both reactive and predictive schedulers should avoid very short periods that reveal the data monitoring and movement costs, as well as arbitrarily long periods that do not allow a prompt response to changes in the data access pattern and create insufficient aggregate data movement.

5.4 System Design of Cori

Design Goals. The objectives of our proposed frequency tuning solution are as follows:

G1 *Bridge the performance gap* left by existing solutions that do not properly tune their page scheduling frequency.

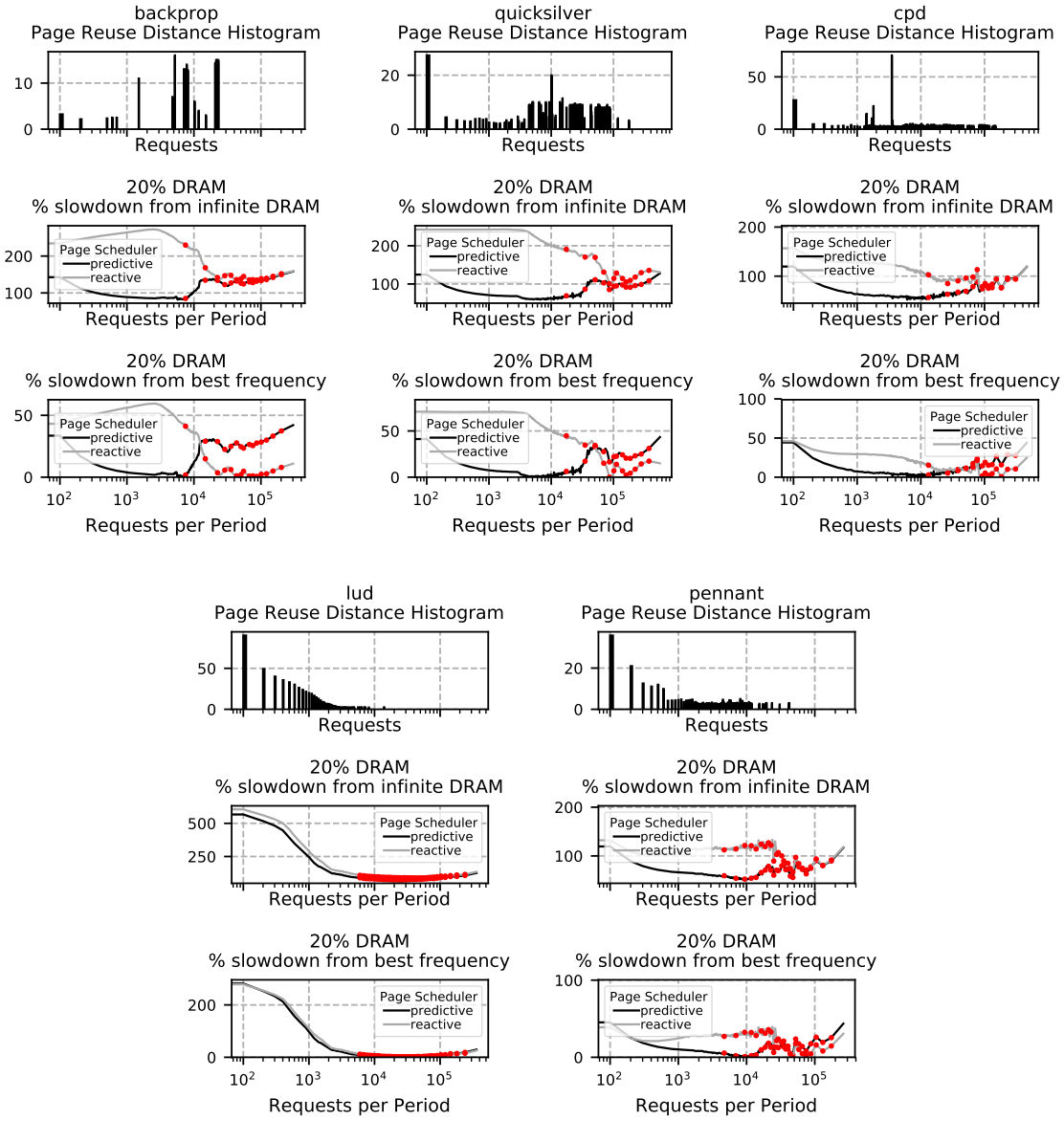


Figure 5.3: Histogram of page reuse distance and its relationship with application performance across period durations, for a predictive and reactive page scheduler over a simulated platform with DRAM and PMEM at a $20\% : 80\%$ capacity ratio. The red dots correspond to the performance of the candidate frequencies generated by Cori.

G2 *Drastically reduce the number of tuning trials* needed to find the frequency that enables desired performance.

G3 *Build a generic tuning approach* that works across applications and page schedulers.

G4 *Enable practical system-level integration* using readily available information on application data access behavior, without explicit code-level modifications or specific APIs.

To address these goals, we propose **Cori**, a method for tuning data movement frequency in hybrid memory systems. Cori gleans data-movement requirements based on application-specific data reuse trends to guide the frequency tuning process, and select a frequency which delivers performance gains or increases in data movement efficiency (**G1**) with a small number of tuning trials (**G2**). Cori extracts the necessary information from execution profiles, and does not require any changes to applications or the memory management stack (**G3**). Experimental results from a real testbed with DRAM and Intel Optane PMEM validate the simulation-bases evaluation of Cori, and demonstrate the feasibility of its system-level integration (**G4**).

Cori Overview. Figure 5.4 illustrates the system design of Cori and its interactions with the hybrid memory page scheduler, summarized as follows:

1. The Reuse Collector executes a single profile run of the application to collect information on data reuse.

2. The Frequency Generator analyzes the data reuse profile and generates a range of proposed data movement frequencies. To achieve this, it first calculates the dominant reuse period as a weighted average of the observed reuses (2a). Then, it generates a range of candidate frequencies at time intervals that are multiples of the dominant reuse period (2b), and outputs the frequencies to the Tuner in decreasing order, from higher to lower frequencies, thus shorter to longer periods.

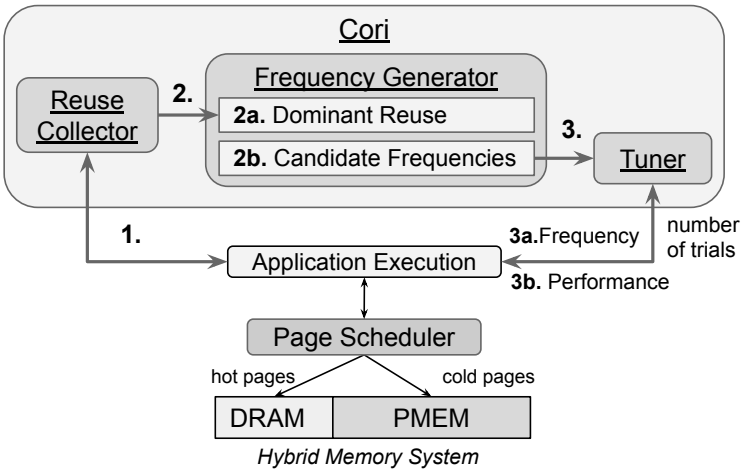


Figure 5.4: System components of Cori and its integration with the hybrid memory software stack.

3. The Tuner makes a number of tuning trials with the candidate frequencies in the proposed order. It configures the page scheduler to operate at each of the recommended frequencies (3a). It then observes the application runtime and resource use and determines whether the application performance has reached best or desired levels (3b). If not, the Tuner moves on to the next frequency in order, going back to step 3a.

Next, we describe in more detail these steps and system components.

Reuse Collector

The goal of the Reuse Collector component is to generate a histogram of data reuse similar to the ones shown in Section 5.2. In the context of the simulation-based analysis we collect memory access traces and have access to detailed information on data reuse in terms of page reuse distances at the granularity of each individual memory access. This cannot be generally achieved for arbitrary applications, therefore, we propose a practical system-level alternative to collect similar information on data reuse.

Loop Durations. We make the intuitive realization that data reuse appears mostly within loop operations during application execution. Therefore, information on the time duration of loops can be a practical estimation to page reuse distance in the time domain. Figure 5.6a

depicts the time duration of loops across applications including `backprop` and `lud`. We observe a similar histogram shape to the ones generated via the memory access traces for the page reuse distances in Figure 5.3: `backprop` has distinct loop durations that repeat around 15 times, which corresponds to the 16 data access strides depicted in Figure 5.2, and `lud` shows a gradual degradation in the loop durations due to the triangular array traversal and decreasing reuse of the number of pages shown in the same figure. We validate that the loop duration histograms of the remaining applications match what we observed via the memory access trace collection.

Collection of Loop Durations. In the context of validating Cori on a native testbed in Section 5.5, we instrumented the applications source code and individually timed the duration of the primary for loops. In principle, however, such instrumentation can easily be performed using compiler-level [70, 71] or binary instrumentation techniques [72, 73]. Currently, we do not present a complete Cori tool which integrates such techniques, rather we focus on establishing the methodology that forms the basis of such a tool, and demonstrate via manual instrumentation that the methodology is effective. We verify that we can obtain accurate loop timings using a LLVM compiler pass, similar to what has been used as part of the Beacons compiler framework [71], which automatically generates the instrumented binary without any application source code modifications.

Frequency Generator

Dominant Reuse. The Frequency Generator analyzes the data reuse histogram provided by the Reuse Collector, in order to identify the one that best represents the range of captured reuses. We refer to this as the *dominant reuse*. Dominant reuse (DR) is computed as a weighted average of the observed data reuses (N different reuses) in the histogram, as summarized in Equation 5.1. The weights are the number of appearances $repeat_i$ of a reuse $reuse_i$ in the corresponding histogram. This will shift the average towards the data reuse distances that repeat more times. Additionally, we introduce an extra weight $(N - i)$

that favors shorter reuse distances, because this will allow us to generate a more calibrated selection of candidate frequencies, that works irrespective of the page scheduler's effectiveness, as we evaluate in Section 5.5.

$$DR = \frac{\sum_{i=1}^{N}(N-i) \times repeat_i \times reuse_i}{\sum_{i=1}^{N}(N-i) \times repeat_i} \tag{5.1}$$

$$CandidatePeriods = [DR,\ 2 \times DR,\ 3 \times DR,\ ...,\ \frac{Runtime}{2}] \tag{5.2}$$

Output Candidate Frequencies. Based on DR, the Frequency Generator creates a sequence of candidate data movement periods at time intervals that are multiples of DR, as shown in Equation 5.2. The last possible candidate in the sequence is the one that splits in half the overall application runtime that the Reuse Collector has previously observed. The candidate frequencies are derived by simply inverting the values of the candidate periods. Figure 5.3 includes a visual representation of the candidate periods as red dots. Finally, the Frequency Generator outputs to the Tuner the candidate frequencies in the specified order from shorter to longer periods, thus higher to lower data movement frequencies. This priority ordering, together with the dominant reuse calculation, is essential to Cori's success, compared to other possible solutions, as we evaluate upon in Section 5.5.

Tuner

The Tuner uses the sequence of candidate frequencies to perform the actual tuning procedure. The Tuner starts its initial trial with the first frequency in order, sets it as the operational page scheduling frequency and executes the application over the hybrid memory. If performance is within desired levels or the best one observed (after the first trial), the Tuner chooses to stop or continue the tuning process. When the Tuner finds the frequency that provides best performance after a number of trials, the selected frequency is kept for any subsequent execution of the particular application on the given combination of

platform configuration and page scheduler.

Discussion

Cori currently improves upon tuning approaches, such as the empirical ones, by observing best performance across a number of tuning trials of actual application execution. The decision of after how many trials the tuning stops is flexible. There can be a fixed number of trials or tuning can stop after performance reaches desired levels or shows no significant variation from the last trial. However, such an execution-based tuning methodology may be impractical for long running applications, such as training machine learning models and scientific simulations. Nonetheless, Cori only requires the collection of data reuse information, that can be made readily available using compiler-assisted instrumention, laying the grounds towards an online frequency tuning solution. Cori can be extended with system-level performance metrics and combined with online access pattern detection solutions used in prefetching [59, 74], or machine intelligent page schedulers [31], so as to adapt the page migration frequency to dynamic changes in data reuse and access patterns. Finally, the recommendations made by Cori depend on the calculation of the dominant reuse, and are therefore sensitive to the granularity at which the data reuse information is collected and aggregated. The evaluations presented later base the calculation on reuse information captured at granularity of 1000s of data accesses (in the simulation framework) and of each loop (on the real hardware testbed). This instrumentation granularity can be dynamically adjusted to trade among the tuning overheads vs. the quality of the recommendations.

5.5 Evaluation

The goal of the evaluation is to demonstrate how Cori realizes its design goals. First, we highlight the benefit of using Cori with respect to application performance improvements and system resource efficiency. Second, we evaluate the tuning overheads of using Cori. Finally, we validate the effectiveness and practicality of Cori on the native Intel Optane

platform.

Benefit of Using Cori

Figure 5.1 includes the application performance and data moved when the page scheduling frequency is tuned with Cori, compared to the frequencies proposed by existing solutions. Regarding application performance, using the frequencies selected by Cori achieves on average a 3% slowdown compared to when using the best possible frequency for each of the applications. In comparison, the frequencies used by other techniques result in an average 10%-100% slowdown from the ideal case. For cases where Cori does not provide the best application performance, as in the case of `quicksilver` with a predictive page scheduler, the performance with Cori is less than 3% away from the best observed one. As discussed in Section 5.2, no other set of frequencies proposed by existing solutions provides as good performance across all applications *and* page schedulers, as Cori.

In this experiment, the frequency tuning in Cori is performed to optimize application performance, so it is not surprising that it does not provide uniformly lowest data movement. However, Cori realizes the necessary data movements to achieve the provided application performance levels. For instance, the increased data movement compared to some of the predictive schedulers (e.g., 3× more compared to thermostat), are offset by the reductions in the slowdown compared to the best frequency case (5×). We highlight the actual number of GBs moved for a native hybrid memory platform later in Section 5.5.

Cori meets the **G1** *design goal by bridging the performance gap left by existing solutions and achieves only a 3% average slowdown from an optimal frequency selection across applications and page schedulers.*

Overhead of Using Cori

We evaluate the overheads of using Cori by comparing the number of tuning trials required by Cori to find the best frequency vs. what is required to find that value using other tuning methods. We also evaluate whether Cori's overheads are justified, by comparing how close Cori is to the performance of a system which operates at the best possible frequency for

each of the applications, vs. how close would the other methods be if they use the same number of trials as Cori.

Given the lack of a non-empirical tuning approach, we construct a **baseline**, which like Cori, operates at the system-level, but is blind to any insights it might have regarding application requirements. This baseline explores the $O(N)$ problem space of all possible frequencies by using a simple step function, with candidate periodic time intervals that differ by a duration of $timestep$, as summarized in Equation 5.3, The corresponding frequencies are derived by inverting the periodic time intervals.

$$Base\,Candidates = [timestep,\ 2 \times timestep,\ ...,\ \frac{Runtime}{2}] \tag{5.3}$$

Next, we vary the **priority ordering** of the generated candidate frequencies. First, the `base-left` baseline starts from low frequencies (large periods) and moves to the left towards higher frequencies (short periods) in the sequence described in Equation 5.3. The `base-right` baseline starts from high frequencies and moves towards the right to lower ones, similar to Cori. Third, we also assume a `base-random` approach that randomly explores values in the sequence.

Figure 5.5a shows the number of tuning trials required for best application performance. Among the baseline variations none of them works best across all applications *and* page schedulers. More specifically, while `base-right` is the baseline approach that requires least trials for a predictive page scheduler across application, `base-left` works best for reactive page schedulers. Thus, a baseline approach that explores frequencies in a certain priority ordering needs further insights to identify best performance in a reasonable number of tuning trials. Even though `base-random` is independent of such a priority ordering, its unpredictable frequency selection results in worst-case average tuning overheads.

In contrast, the guided frequency selection performed by Cori, allows it to make a recommendation in just two trials on average for predictive page schedulers. Across appli-

cations and page schedulers Cori reduces the number of trials by **5**×, from 25 on average across baselines down to only 5 trials. The only corner case where Cori requires up to 20 trials is for applications with random access patterns like `bfs` and `bptree` where the access pattern prediction capabilities of a reactive page scheduler are limited. This is the reason why for such applications, when a more predictive page scheduler [31] is not available, the cache organization of the hybrid memory is shown to be more beneficial [46].

Cori meets the **G2** *design goal by reducing by 5× the number of tuning trials needed to reach an average of only 3% performance slowdown, compared to baselines that ignore insights about application requirements, and as low as 2 trials on average for predictive page schedulers.*

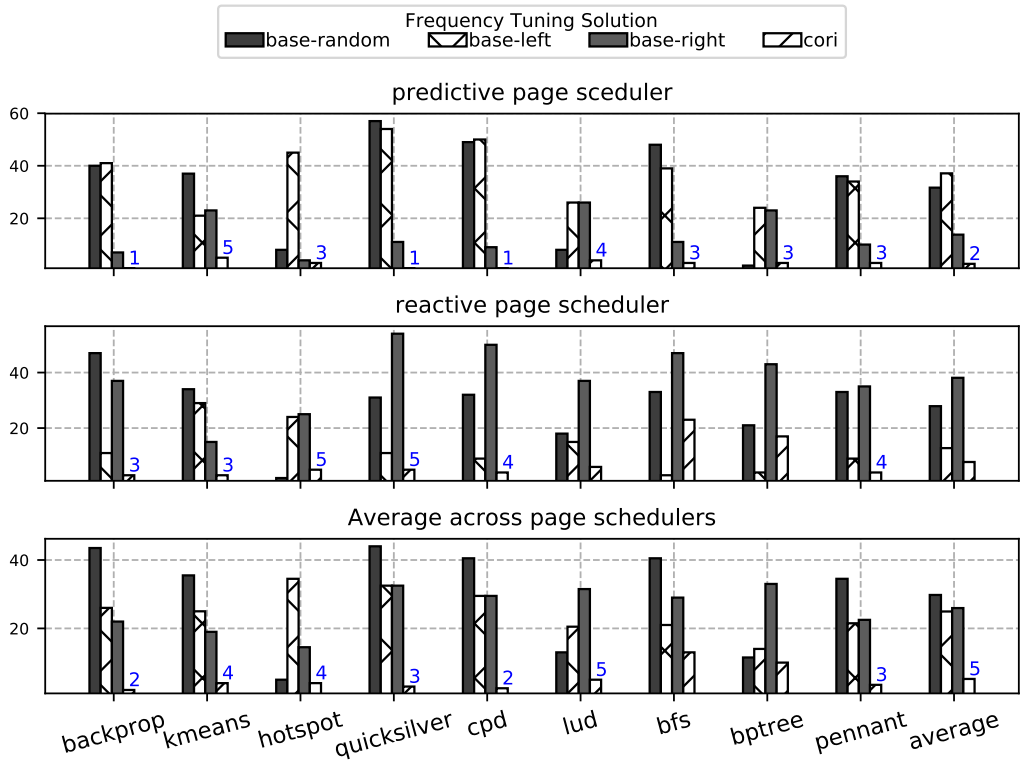
Figure 5.5b shows the performance that the baselines provide when executing for the same number of tuning trials that Cori requires to find best performance. The values are averaged across the page schedulers. On average, the baselines incur higher performance slowdown because they require significantly more trials to reach best performance, as shown in Figure 5.5a. Within the execution overhead of Cori, only the `base-random` approach seems to be able to still choose frequencies that provide good performance, but only for some of the applications; for others (e.g., `quicksilver` and `pennant`), `base-random` is less effective even compared to some of the other baselines.

For completeness, in Figure 5.5c, we also show the best frequencies selected by Cori for the two types of schedulers. We highlight that the best frequency that maximizes application performance differs across schedulers and across applications, further justifying the use of Cori.

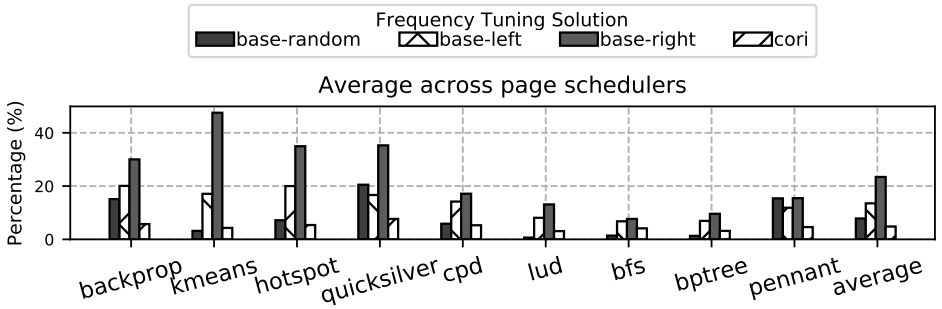
Cori meets the **G3** *design goal since it provides maximum performance improvements for minimum number of tuning trials across applications and page schedulers.*

Optane DC PMEM Validation

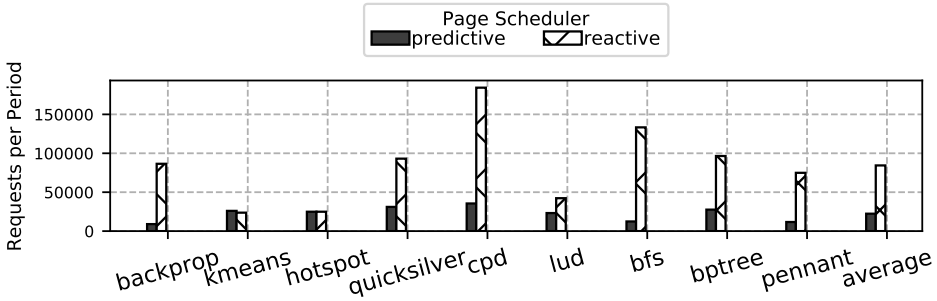
We validate the simulation-based observations about Cori by providing results from a native hybrid memory platform. These experiments also demonstrate the feasibility of using Cori



(a) Number of tuning trials to find best performance. Cori (blue text) requires the minimum number of trials on average across all applications *and* page schedulers.



(b) Performance slowdown from best frequency for Cori's number of trials. Cori is the only solution that provides lowest slowdown consistently for *every* application *and* page scheduler.



(c) Final period time duration selection of Cori.

Figure 5.5: Comparison of Cori with a baseline frequency tuning solution for a simulated hybrid memory system with DRAM and PMEM at 20%:80% capacity ratio.

as a practical system-level solution for frequency tuning. The experiments are conduced on an Intel Optane platform, configured with 20%:80% DRAM to PMEM capacity ratio, and a reactive page scheduler kernel module that operates over a window of past access history, both as described in Chapter 3. Then, we go through the steps of Cori as summarized in Figure 5.4 and report our findings in Figure 5.6.

Recreating Cori's steps. First, we gather information on data reuse. More specifically, we collect the time duration of the loops across applications, as shown in Figure 5.6a, using the suggested approach in Section 5.4. Second, we calculate the dominant reuse as described in Equation 5.1 and generate the candidate period durations at multiples of the dominant reuse, as shown in Figure 5.6b. While for `backprop, kmeans, hotspot` the dominant reuse is around 1 sec, for `lud` it is much less, given the corresponding loop duration histogram. We also include period durations that are less than the dominant reuse, to validate whether performance indeed is not best for such periods that Cori does not include in its sequence of candidates. Third, we replicate Cori's tuning process by executing the applications for the selected period durations in increasing order and observe the runtime, its slowdown from the ideal case of infinite DRAM and data moved, as shown in Figure 5.6c. The final choice according to Cori is the *first* period duration in the experimentation order that drastically reduces the amount of data moved and thus appropriately reduces the runtime. Figure 5.5c inidicates in blue the final period choice and the number of tuning trials it required.

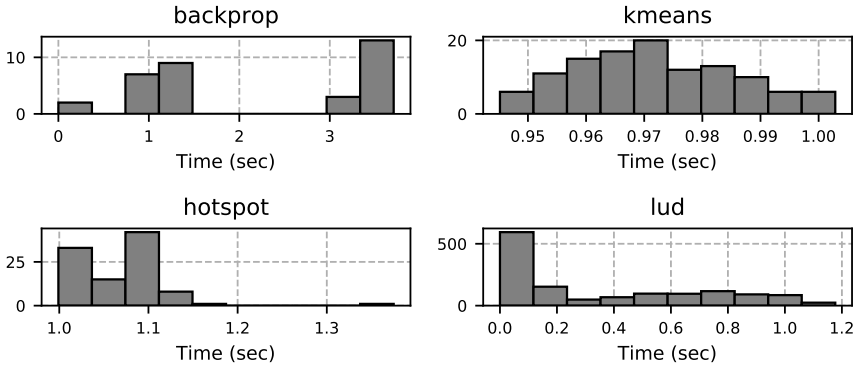
Validation observations. First, we observe that period lengths that are shorter than the dominant reuse (DR/4, DR/2), create tens of GBs of more data moved, consistently across all applications. This confirms the insight presented in Section 5.2 that the operational period should not be shorter than the data reuse pattern. Also, it validates Cori's effectiveness in calculating the dominant reuse and choosing it as the initial point of tuning. The performance with much larger periods is not included in Figure 5.6c, since it can be substantially worse, such as 50% of runtime slowdown for `lud` at 5 second periods, and Cori's tuning

ends at much shorter periods.

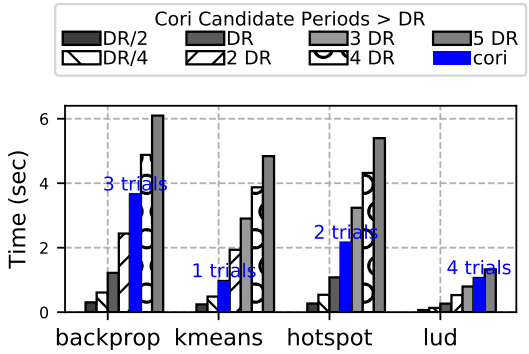
Second, regarding application performance and system resource efficiency, Cori selects the period duration that reduces to their lowest levels both the data moved and the runtime slowdown from the case of infinite DRAM capacity, across all applications. For some applications these levels of runtime slowdown are less significant than others. For applications like `kmeans` and `lud` very short periods that force the reactive page scheduler to create a burst of asynchronous data movements, are not enough to stress the Optane's bandwidth and proportionally reflect on their runtime. Regardless, Cori identifies the page scheduling frequency that enables the best performance levels allowed by the available DRAM capacity and minimizes data movement overheads. Additionally, the levels of runtime slowdown observed in this experiment, are very similar to the ones captured in our simulation (Figure 5.3), validating its correctness.

Finally, the selected periods themselves are different across applications and range between 1-3 seconds. Even though this doesn't seem as a substantial difference, empirical approaches may have ignored values in such proximity, however, for `backprop` the runtime slowdown reduces by 50% when going from 1 second to 3 second periods, and for `hotspot` by 30% when switching from 1 second to 2 second periods. This validates the benefit from using Cori toward realizing significant application performance improvements, within only 2-3 average tuning trials, minimizing the tuning overheads.

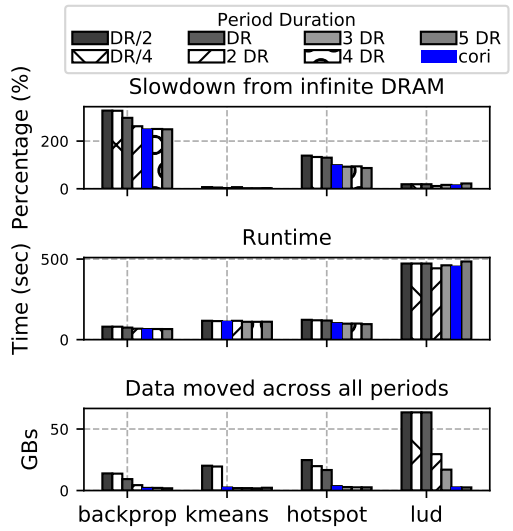
Cori meets the **G4** *design goal by allowing for a practical integration with existing hybrid memory system-level managers, and can be realized without modification to applications and system-level components. Validation of Cori on the Intel Optane DC PMEM platform, confirms the simulation-based motivational arguments and insights, and highlights the benefit of using Cori in return for minimal tuning overhead.*



(a) **Cori Step 1:** Collect loop time durations.



(b) **Cori Step 2:** Calculate the dominant reuse DR and generate candidate period lengths at multiples of the DR. **Final Choice:** Select the first period length (blue bar) after x trials, that brings lowest runtime *and* migrations.



(c) **Cori Step 3:** Tuning trials of application performance.

Figure 5.6: System-level validation of Cori for a reactive page scheduler that executes on a native Optane DC PMEM platform for 20%:80% DRAM:PMEM capacity ratio.

5.6 Chapter Summary

This chapter presented Cori, a system-level solution for tuning the operational frequency of data tiering solutions that periodically move data across flat hybrid memory components. Cori reveals that related works do not properly configure their operational frequency relying on empirical tuning, thus failing to deliver up to 100% of performance improvements. Cori synthesizes insights on data reuse information to better guide the process of selecting frequency candidates, reducing by, on average, 5$\times$ the number of tuning trials from an insight-less exploration. This way, Cori delivers performance improvements within 3% from the case of optimally chosen frequency. Cori is robust, and provides benefits across application data access patterns and page migration policies. Such benefits are complementary to the use of machine learning and further boost its effect on application performance.

CHAPTER 6

SCALING MANAGEMENT OPERATIONS WITH PATTERN CLUSTERING

The previous chapters describe solutions that enable machine learning-based predictions (Kleio), and fine-tuned operation (Cori), and deliver significant performance improvements over current hybrid memory management systems. The reminder of this thesis focuses on how to reduce the operational overheads of machine learning-based hybrid memory managers, like Kleio, and strengthen the foundations for their practical use as system-level solutions. In this chapter, we identify that even though Kleio selects a small page subset for machine learning-based management, the size of the subset can vary across workload sizes and patterns, resulting in significant learning overheads. To this end, this thesis contributes **Coeus**[1], a page clustering mechanism that enables the management of more pages under a single machine learning model, thus reduces the aggregate number of models and the associated training costs. In this chapter, we reveal the limited effectiveness and practicality of using well known data clustering methods. In contrast, we leverage the data reuse insights described in the previous Chapter, to increase pattern similarity and facilitate lightweight page grouping.

6.1 Overview

In Chapter 4 we presented Kleio, a hybrid memory page scheduler with machine intelligence. Kleio lays the foundations for the practical use of machine learning by identifying a small page subset (small with respect to the overall memory footprint of the application) that benefits from ML-based management. However, the absolute number of the pages managed with machine intelligence can vary substantially across workloads. It depends on various parameters, such as the aggregate number of pages and type of access patterns.

[1]The name is inspired by ancient Greek mythology, where Coeus was the titan god of intelligence.

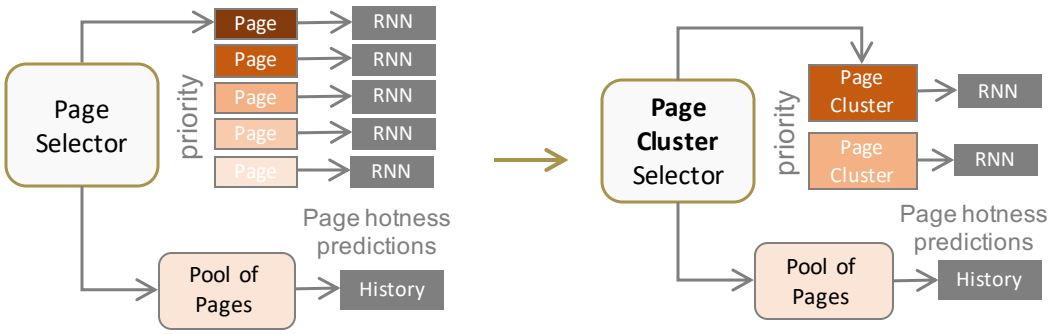


Figure 6.1: Scaling machine learning models to learn patterns across a page cluster instead of a single page.

Therefore, certain applications may need significantly more resources for training than others. To realize a practical system-level integration of solutions based on machine learning effectively across application classes and inputs, it is necessary to further reduce the resource requirements and overheads associated with machine learning.

The most intuitive approach in reducing the overheads of deploying per page models is to create fewer models that correspond to more pages by clustering pages and training a single model per page group, as depicted in Figure 6.1. Yet, this is not a trivial task, as it introduces new design questions and challenges. For instance, which, how many and with what criterion should pages be grouped in the same cluster? What is the optimal number and composition of clusters for the purpose of training a single RNN per cluster with high prediction accuracy and low training times? The use of unsupervised machine learning clustering methods can potentially resolve some of these questions. However, these come with substantial execution overheads and configuration constraints, such as knowing a priori how many clusters to create. We ask then; *Can we build a fast and robust approach that allows machine learing models to be associated with a larger number of pages – as what could be achieved with clustering – without the additional complexity that use of unsupervised learning techniques would introduce?*

Contributions. This part of the thesis proposes **Coeus**, a page grouping mechanism that facilitates the practical integration of machine learning in system-level hybrid memory man-

agement. Coeus bypasses the complexities and overheads of running well-established data clustering methods. Instead, Coeus leverages data reuse insights to analyze the training data and to then instantly create efficient clusters of pages. The specific contributions are the following:

- We demonstrate that the resource requirements for training a single machine learning model per page, for the page subset selected by Kleio, vary up to $9\times$ across applications with different memory footprints and data access patterns (Section 6.2).
- We explore widely used machine learning data clustering methods to group pages together, so as to train a single machine learning model per page cluster. We argue that these methods introduce additional overheads and complexity to the already complex hybrid memory management pipeline (Section 6.3).
- We leverage insights on data reuse times to group pages together at no additional overhead, while ensuring efficient RNN deployments (Section 6.3).
- We describe the design of Coeus and how it integrates with existing machine intelligent hybrid memory management solutions, such as Kleio. (Section 6.4). The implemented code base is open sourced[2].
- We evaluate Coeus against Kleio and show that it reduces the associated learning overheads by almost $3\times$ and increases application performance by $3\times$ (Section 6.5).

6.2 Motivation

Kleio's original evaluation over 20 different workloads, given the substantial amount of resources required and the available experimental setup, allowed for training RNNs for only 100 pages per workload, as summarized in Section 4.6. Figure 6.2 shows how application performance increases (y-axis) the more RNNs are deployed (x-axis) compared to the ones

[2]https://github.com/GTkernel/coeus-sim

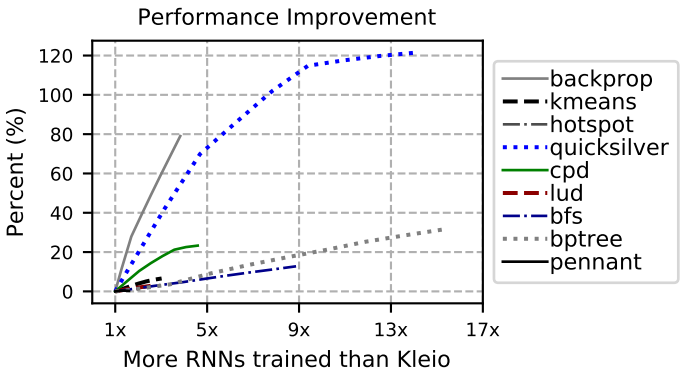


Figure 6.2: Application performance improvements across larger number of per page RNNs deployed from the ones allowed by the available rersources in Kleio's evaluation ($1\times$).

used in Kleio's evaluation. We observe that certain workloads, such as `quicksilver` require at least $9\times$ more RNNs to be trained, to reach maximum performance improvements, which are 120% higher than the one provided with Kleio's evaluation. Similarly, the performance of `backprop` improves by 80% and `cpd` by 20% with $4\times$ more RNNs than Kleio. The non-linear increase in performance of these workrloads is inherent to the page priority ordering that Kleio follows, as mentioned above. Finally, workloads with more random access behaviors, such as `bptree` and `bfs`, linearly improve their performance by 20% for $9\times$ more RNNs than Kleio.

The variability in the performance improvements across larger number of RNN trainings, is inherent to application-level characteristics regarding their data input sizes and data access patterns. Figure 6.3 shows on the x-axis the size of the memory footprint for the data input sizes configured in our experimental setup. The scale is relative to the smallest memory footprint we observe across workloads, that is the one corresponding to `hotspot`. This characteristic is one parameter contributing to the number of per page RNNs needed to achieve maximum performance improvements, which is particularly high for `quicksilver`, `bptree` and `bfs` that have larger number of pages compared to the rest of the workloads.

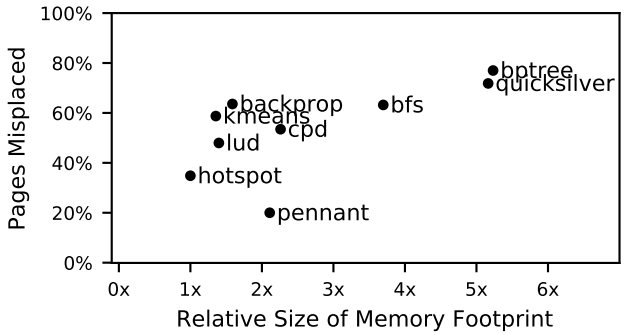
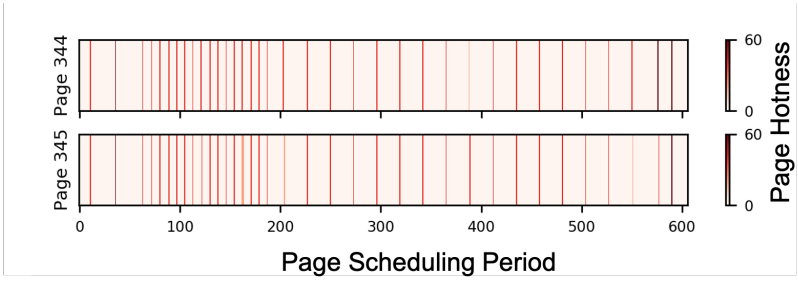


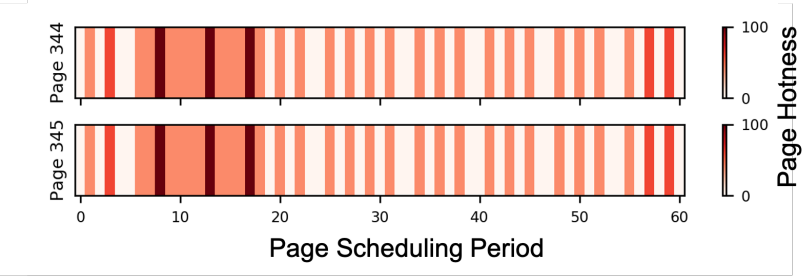
Figure 6.3: Workload characterization with respect to the relative size of their memory footprint for the selected data inputs (x-axis). Percentage of page misplacements by history-based page schedulers (y-axis).

Apart from the aggregate memory footprint, the other application-level characteristic that contributes to the number of RNNs that need to be trained, is the complexity in the access patterns. Applications with higher complexity in access behavior have higher need for machine learning-based management, because history-based management fails to effectively capture more intricate patterns. Kleio's Page Selector identifies such complex patterns by observing which pages are frequently misplaced by simple history-based predictors. The y-axis of Figure 6.3 shows the percentage of such misplaced pages from the overall workload's memory footprint. We observe that workloads with more complex, yet structured access patterns, such as `quicksilver`, have 70% of their memory footprint be misplaced, thus mis-managed, by history page schedulers, thus would benefit from the RNN training of this page subset. Similarly, in the case of `bfs` and `bptree` that exhibit irregular and random accesses, they also require more than 60% of their footprint to be trained with RNNs.

Takeaways. The resource requirements for machine learning-based hybrid memory management can vary significantly across workloads. This is inherent to the workload's memory footprint and access patterns, as well as to the design choice of training a single RNN model per application page out of a insight-fully selected page subset, whose size varies across workloads.



(a) Page access patterns are *similar* over very short page scheduling periods.



(b) Page access patterns become *identical* over larger page scheduling periods.

Figure 6.4: The memory access patterns, that machine intelligent page schedulers learn, are the per page access counts across the scheduling periods.

6.3 Clustering Similar vs. Identical Patterns

The results in the previous section show that the deployment of a single machine learning model per application page is not scalable with respect to resources required for training across workload classes and inputs. To reduce the aggregate learning overheads and resource utilization it is necessary to increase the granularity of the address space managed under a single model. Our approach is to train individual models at the granularity of a page cluster, and learn the specific access patterns of the pages in the cluster.

Machine intelligent page schedulers, like Kleio, learn patterns across *time*, not space, training a single RNN per page. These patterns correspond to the sequence of page access counts (hotness) across the page scheduling periods. Figure 6.4a shows a heatmap of the page hotness for two neighboring pages of the workload `backprop`. The hotness across the two pages seems to be very similar, with few differences in the absolute access counts for specific periods. Since their patterns are so *similar*, it is intuitive that these

two pages should belong in the same page cluster and correspond to the same RNN deployment. Therefore, we need a mechanism that identifies similarities across page access patterns and creates such groups of pages. The **challenge** in this approach is to execute the page clustering incurring trivial overheads. The machine intelligent hybrid memory management already involves substantial overheads and complexity, which should not be further exacerbated with an intricate clustering process.

Therefore, we first try to use unsupervised machine learning data clustering methods that are widely used across domains. In particular, they are very effective in forecasting time-series (patterns in time), that are grouped according to their similarity [75, 76, 77]. However, we make a case that it is particularly challenging to optimize upon the number of clusters created, while the process introduces additional overheads and configuration constraints. Then, we show how to bypass such complexities and use data reuse insights to instantly create large page groups of *identical* access patterns, which can be inferred using a single input to a single machine learning model.

Clustering Similar Patterns

Overview of K-means. First, we explore widely used machine learning methods to group together data that share similar characteristics. Common terminology refers to the input data as *observations* or *samples*, that each have a set of *features*, i.e., characteristics. The similarity between the observations is referred to as *distance*, which involves some algebraic calculation of the difference between the values of the individual features across observations, e.g., euclidean distance. One of the most commonly used clustering algorithms is k-means, which partitions observations into a predefined number of `k` clusters. The algorithm works by randomly selecting `k` observations as *centroids* and groups together observations with a small distance from these centroids. The algorithm repeats and optimizes the centroid selection, so that the sum of squared distances from the centroid, known as *inertia*, minimizes.

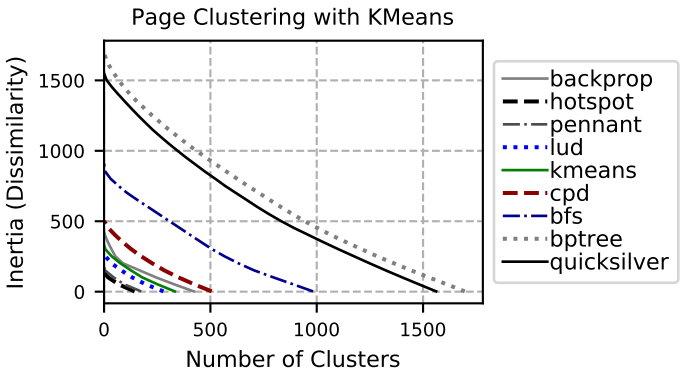


Figure 6.5: Page cluster inertia (dissimilarity) across increasing number of clusters created with k-means, for the page subset selected for machine learning. Larger number of clusters include fewer pages that are more similar.

Number of clusters. K-means requires to define the number of clusters before execution. Yet, setting this value is not always intuitive for a given dataset, whose characteristics we are not aware of. According to a thorough survey of all clustering methods, finding the optimal number of clusters and the similarity metric are the two biggest challenges and constraints in clustering [78]. This is why other methods such as hierarchical clustering, have higher time complexity, since they create a hierarchy of all possible number of clusters. One way to find the optimal number of clusters `k` for k-means, is to run the algorithm for a reasonable range of `k` values and select the one, after which clusters don't change drastically. In a more formal mathematical description, this is described as the *elbow method* or otherwise known as finding the 'knee of a curve'. In particular, for k-means the curve corresponds to how the *inertia*, i.e., the sum of squared distances of observations to their cluster's centroid, decreases as we increase the number `k` of clusters. Lower inertia means higher similarity within a cluster. **Zero inertia** corresponds to clusters of data samples with identical feature values. A good selection of `k` number of clusters is the one where inertia doesn't drastically reduce, i.e., 'shows a knee', if we create an additional cluster.

Application of K-means. In the context of page clustering, the input observations are the different application pages and the per page features are the page access counts across

the scheduling periods, as depicted in Figure 6.4. This feature selection allows k-means to cluster together pages that received similar number of accesses across periods. Prior to clustering the features are normalized between 0 and 1, as it is a common practice in machine learning.

Complexity in the Configuration of the Clustering. Figure 6.5 shows how the dissimilarity of the pages in a cluster decreases, the more clusters we create, thus the fewer pages a cluster contains. The lower the inertia the more similar are the pages within the cluster. When inertia is zero the pages of the cluster are identical, meaning they receive exactly the same access counts across scheduling periods. The clustered pages are the ones selected for RNN training by the machine intelligent page scheduler. The selected page subset contains pages whose patterns cannot be accurately predicted with history-based approaches, thus need machine learning-based predictions.

In general, we observe that the inertia decreases very slowly the more clusters we create, almost in a linear way. There is not a distinct number of clusters where the inertia curve significantly dips, in other words there is no 'knee'. Therefore, it is not obvious which number of clusters is best to choose even after experimenting with all possible cases, let alone defining it prior to k-means execution, as required. This behavior is inherent to the patterns of the pages selected for machine learning. These patterns are highly dissimilar and this is exactly why they need machine learning-based management. We also observe that empirically selecting a number of clusters that could work well for one application, may not be as effective for another. For example, creating 500 clusters is best for `cpd`, but results in highly dissimilar clusters for `bfs`, `quicksilver` and `bptree`. Therefore, the number of clusters needs to be tuned on an application basis.

Additional Complexity in the RNN deployment. Another design aspect of clustering based on similarity, that is not straight-forward, is how to train a single RNN model across inputs of different pages. There are two possible design choices. The RNN model can take as input the access patterns of all pages. Therefore, now the input grows linearly to the number of

pages in the cluster. This will increase the training time of a single RNN model, as it has to manipulate bigger input, learn more patterns and possibly take longer to converge. Arbitrary increase of the cluster size may lead to prohibitive training times, which is the primary reason why machine intelligent page schedulers create models at the granularity of the page. To avoid this increase, which contradicts the purpose of this work, the input to the RNN model can be the access pattern of a single page of the cluster. This page should correspond to the *centroid* of the cluster, which has highest similarity with the rest of pages of the cluster. However, according to the size of the cluster and how similar pages are, there may be the case that the RNN model that is trained on the centroid, does not make highly accurate access patterns predictions for the rest of pages of the cluster. This can particularly be the case since the clustering is applied across pages which have already been identified as ones which need machine learning to aid with page scheduling, and thus are more likely to have dissimilar patterns. Therefore, we ideally need a clustering mechanism that does not increase the per model training times and does not compromise on the prediction accuracy.

Takeaways. The use of machine learning clustering methods for the purpose of page clustering introduces new complexity in the already strenuous problem of hybrid memory management. The complexity comes from the fact that is not intuitive how many page clusters to create, given how dissimilar the target pages are, and how to best configure the training of a single RNN model for input of more than one page with minimal overheads. Ideally, we seek a simple technique to completely bypass all such complexities, leveraging insights and observations that will allow for instant creation of large clusters of pages that do not require any changes to the way RNNs are deployed.

Clustering Identical Patterns

We next make the observation that it is trivial to calculate the number of page clusters created with zero inertia, i.e., where pages are identical. This is the point where the curve hits the x-axis in Figure 6.5 and corresponds to the case where each cluster contains data

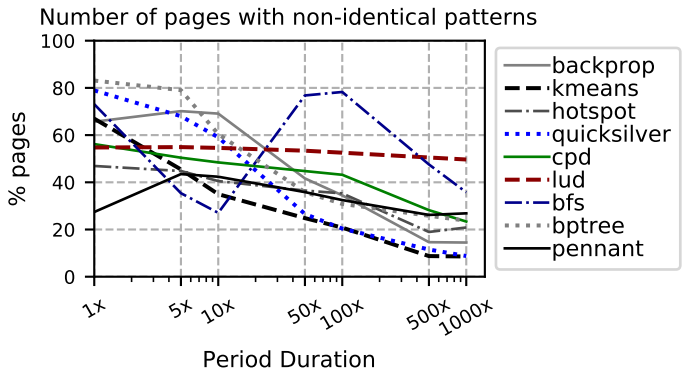


Figure 6.6: Number of pages with non-identical patterns as percentage of per application pages across longer periods compared to Kleio's ($1\times$) period duration. Fewer pages means fewer and larger clusters with identical pages.

samples whose features have exactly the same values. This can be done with a single comparison of the per page access patterns to see if they already belong to the global set of patterns. In this way, we can instantly create clusters of pages and completely *remove the complexity in the configuration of the clustering* as described in the previous section. In addition, the fact that patterns of a cluster are identical, also *removes the complexity in the RNN deployment*. The input to the RNN now can be the access pattern of a single page of the cluster, which does not require any increase in the training times. Since each page of the cluster shares the exactly same pattern, this trivially preserves the same prediction accuracy as when the model has been trained on the input of each page individually.

The challenge that still remains is that the number of clusters with identical pages varies significantly across workloads and can be significantly large given how dissimilar pages are, and the workload's memory footprint and patterns. Is there a way to create fewer and larger clusters of pages with identical patterns?

Looking back at Figure 6.4 we see how the page access patterns look in the context of machine intelligent page scheduling. The difference between Figure 6.4a and Figure 6.4b is the number of page scheduling periods. A page scheduling period determines not only when page scheduling operations are performed, but also the granularity at which page

access data (or data traces) can be used as inputs to an RNN model. This impacts the length of the pattern and its absolute values. Longer periods create patterns that span a smaller number of periods and have larger page hotness values, since more page accesses are received at a longer time interval. When zooming out and observing page access counts over larger periods, the access patterns of the same pages transform from *very similar* (Figure 6.4a) to completely *identical* (Figure 6.4b).

Observing patterns at the right granularity. Given these insights, observing patterns at a more coarse-grain granularity, that is the duration of the page scheduling period, increases the similarity of certain page access patterns to be completely identical. Figure 6.6 shows how the number of page clusters with identical patterns decreases, as the page scheduling periods become longer in time. The period duration selected by related works, such as 0.01 seconds in Kleio, is too fine-grained and creates dissimilarities in the patterns, as depicted in Figure 6.4a. Periods that are $10\times, 100\times$ and $1000\times$ longer than Kleio's, create fewer and larger page clusters with identical patterns for majority of the workloads evaluated. Related memory management solutions [12, 14, 28, 13, 15] adopt such values of data monitoring and page scheduling intervals, that span across orders of magnitude. The decrease in number of clusters with identical patterns is prominent for workloads that have certain structure in their patterns, even if these are simple access strides, such as `backprop`, `kmeans` and `hotspot` or more complex ones, such as `cpd` and `quicksilver`. For these workloads, zooming far out at very long periods, creates fewer and larger clusters of identical patterns, whose number corresponds to only 10% - 30% of the workload's memory footprint. This is not necessarily the case for applications with more irregular data access behavior. For instance, the triangular matrix traversal of `lud` and high accesss randomness `bfs` result in page access patterns that are dissimilar, no matter the length of period. In conclusion, longer page scheduling periods enable the observation of the patterns of page access counts across periods at a granularity that allows the creation of larger and fewer page clusters with pages of identical patterns for applications with structured access patterns.

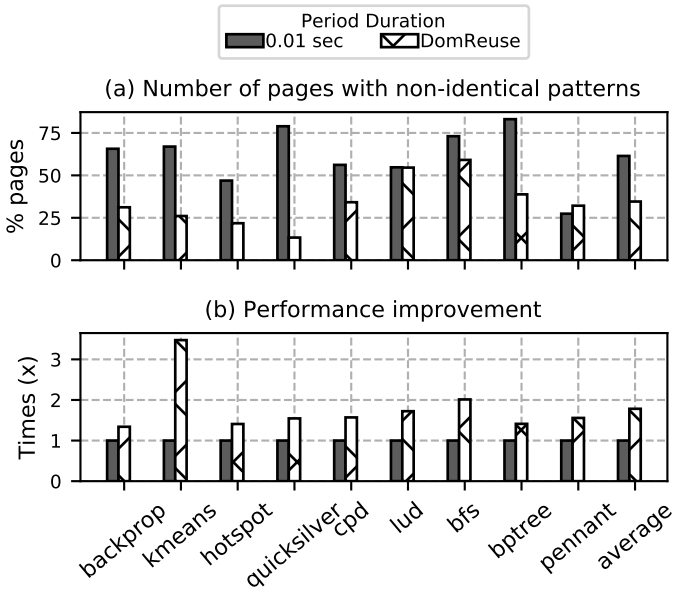


Figure 6.7: Number of pages with non-identical patterns (a) and application performance improvement (b) for history-based management over the period duration selected by Kleio (0.01 sec) and the one (DomReuse) calculated by Cori.

Which is the right granularity? However, changes in the duration of the page scheduling period have a direct impact on application performance, since it affects the timeliness and frequency of data movements across the hybrid memory. Arbitrarily increasing the length of the period may result in insufficient data movements that do not respond in time to changes in the workload's memory access behavior. We need to set the period duration such that it is large enough to create few and large page clusters of identical patterns, without compromising on performance. To this end, our own previous work builds a practical system-level tuning solution – Cori [32] – that finds the page scheduling period duration which maximizes application performance over hybrid memory systems. Cori uses readily available information on page access behavior to extract the per page reuse times and synchronizes the page scheduling periods with the average reuse times. More specifically, Cori calculates the weighted average of the reuse times that are 'dominant' across pages, that is the *dominant reuse* time. Experimental evaluation of Cori over simulation infrastructure and validation with an actual hardware platform with DRAM and Intel Optane persistent

memory shows that hybrid memory page schedulers which operate at a period duration that matches the per workload's dominant reuse time, provide maximum levels of application performance.

Figure 6.7 shows the relation between the number of pages with non-identical patterns and performance when we set the page scheduling period as the dominant reuse time, compared to the 0.01 seconds that Kleio has chosen in its original configuration. Regarding application performance (Figure 6.7(b)) we see that the choice of dominant reuse as period provides almost $2\times$ performance improvements, on average, across workloads. This is for the case when a history-based scheduler is used, therefore we expect that a machine intelligent scheduler will also benefit, as we will see in Section 6.5. At the same time, the period selection results in 30% fewer pages with non-identical patterns, on average across workloads, compared to Kleio's period selection, as shown in Figure 6.7(a). This essentially means 30% reduction in the total number of clusters that have pages with identical patterns. The reduction can be as drastic as 60% for workloads like `quicksilver`, which have structured yet complex access patterns. However, as previously mentioned it is not as significant for workloads with irregular or random access patterns. In conclusion, using the period duration proposed by Cori, with trivial calculations over readily available page access information, allows the observation of the patterns at a granularity that facilitates the creation of large page clusters with identical patterns, while also maximizing application performance.

Takeaways. Observing page access patterns at a coarse-grain granularity increases their similarity to being completely identical. Clustering pages with identical access patterns allows us to bypass complexities and overheads of applying machine learning data clustering methods. The use of insights on data reuse times allows the instant calculation of the page scheduling period duration that enables the creation of fewer and larger page clusters with identical patterns, while ensuring high application performance levels.

6.4 System Design of Coeus

Design goals. The objectives of the proposed solution are as follows:

G1 *Reduce the aggregate number of RNNs* that need to be deployed to enable maximum application performance improvements. In other words, manage more pages intelligently with a certain number of RNNs, compared to current machine intelligent solutions that deploy a single RNN per page of a specific page subset.

G2 *Deliver higher application performance* when training the same number of RNNs as current machine intelligent solutions.

G3 *Introduce minimal operation overhead* in the machine intelligent hybrid memory management process.

To address these goals, we propose **Coeus**, a page grouping mechanism for machine intelligent page schedulers over hybrid memory systems. Coeus creates clusters of pages with identical patterns of page access counts across the page scheduling periods, leveraging data reuse insights to increase the pattern similarity. Then, Coeus identifies the patterns for which RNNs should be trained. In this way, Coeus enables the machine intelligent management of more pages (**G1**), by using a single RNN per pattern, thus page group. In addition, Coeus drastically improves application performance (**G2**), not only due to the intelligent management of more pages, but also due to the fine-tuned page scheduling frequency, that current machine intelligent page schedulers, such as Kleio [31], had missed to achieve. Finally, Coeus leverages the same input with such solutions and runs at a trivial overhead (**G3**), since all of its actions take trivial time to complete.

Coeus Overview. Figure 6.8 illustrates the system design of Coeus and its interactions with a machine intelligent page scheduler for hybrid memory systems.

1. The Pattern Analyzer takes as input the memory access trace and extracts a heatmap

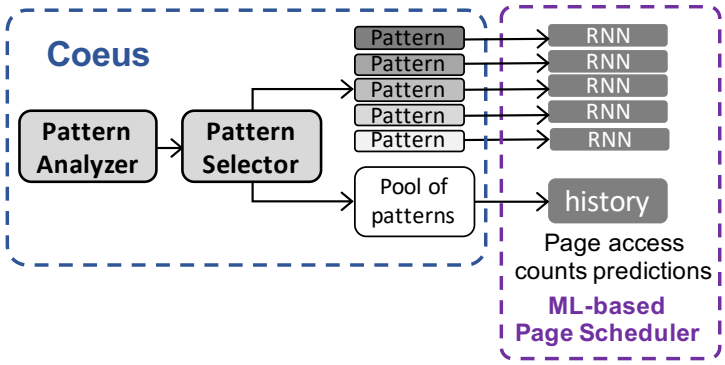


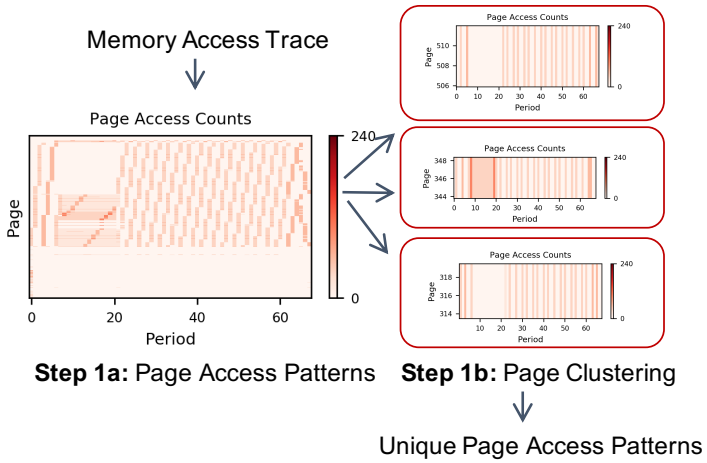
Figure 6.8: System design of Coeus and its interaction with a machine intelligent page scheduler.

of per page access counts across page scheduling periods, calculating the dominant page reuse distance and using it as the period duration (Step 1a). Then, it identifies the page access patterns that are unique across pages, creating clusters of pages with identical patterns (Step 1b).

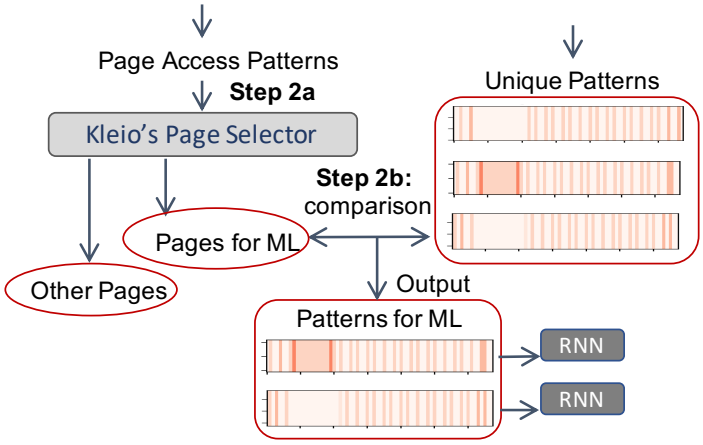
2. The Pattern Selector runs Kleio's Page Selector 'as-is' using the page access count heatmap from the Pattern Analyzer (Step 2a). Then, the Pattern Selector chooses the unique patterns that are shared across pages selected for machine learning-based management (Step 2b). The final output are these unique patterns that are the input for training a different RNN model per pattern, in parallel.

Next, we describe in more detail the internal functionality of the system components.

Pattern Analyzer. Figure 6.9a shows the pipeline of the Pattern Analyzer component of Coeus. First, the Pattern Analyzer converts the memory access trace into a heatmap of page access counts across page scheduling periods (Step 1a). To make this conversion the Pattern Analyzer first sets the period duration to be the length of the application's dominant page reuse distance, as described in Section 6.3. This distance is calculated using information that is already available in the memory access trace and the analytical formula proposed by Cori [32]. Kleio extracts similar page access count heatmap, but arbitrarily sets the page scheduling period duration at 0.01 seconds across workloads. Next, the Pattern Analyzer



(a) Pattern Analyzer.



(b) Pattern Selector.

Figure 6.9: System components of Coeus.

isolates the sequences of page access counts across periods (page access patterns) that are unique across pages. This process is equivalent to creating clusters of pages with sequences that are completely identical, meaning that every access count across periods is exactly the same for the pages of the cluster (Step 1b). Finally, the Pattern Analyzer outputs the set of unique page access patterns.

Pattern Selector. Figure 6.9b shows the internal functionality of the Pattern Selector component of Coeus. The input to this component is the heatmap of page access patterns and the unique patterns that the Page Analyzer identified. The Pattern Selector runs Kleio's Page Selector component 'as-is', taking advantage of its clever selection of which pages

to prioritize for RNN training. The output of Kleio's Page Selector are two different page subsets. One corresponds to pages that should be managed intelligently and the other to the rest of the pages which are efficiently managed by lightweight history-based page schedulers. At this point Kleio would proceed with training RNNs for as many pages as possible out of the corresponding subset. In contrast, Coeus compares the patterns of the pages selected for machine learning with the unique patterns provided by the Pattern Analyzer. In this way, the Pattern Selector returns which *patterns*, not *pages*, should be prioritized for RNN training, following the priority ordering of the corresponding pages.

Interaction with Page Scheduler. Coeus provides the unique page access patterns for which a machine intelligent page scheduler should deploy RNNs for offline training and online inference. In addition, Coeus provides the page scheduling period duration that will improve upon the scheduler's performance, as described in Section 6.3. During application execution, upon every page scheduling period, the page scheduler will use the same RNN to infer the access counts of all pages that share the same pattern, belonging to the same page cluster, using as an identity matching the pattern itself. In this way, the page clustering allows for higher number of pages to be managed intelligently compared to the number of RNNs deployed. Coeus does not intervene in the page scheduling itself, that is the selection of which pages to move. The page clustering of Coeus is not used to schedule groups of pages, only to facilitate the accurate prediction of page access counts that the page scheduler employs to decide which pages to periodically move across hybrid memory.

6.5 Evaluation

The goal of the evaluation is to demonstrate how Coeus realizes its design goals, as described in Section 6.4. We highlight the application runtime performance and reduction in machine learning overheads that Coeus enables. Second, we enumerate the execution overheads of Coeus and argue that they have trivial impact on the ML-based page scheduling overheads. The experimental evaluation is done over the hybrid memory simulation

environment, performance estimates and applications described in Chapter 3.

We evaluate Coeus against Kleio, since it is designed to optimize upon the initial design of Kleio. However, for the evaluation of Coeus, we do not go through an actual deployment of Recurrent Neural Networks, since we have evaluated the achieved prediction accuracy levels in Section 4.6. Since Coeus inputs *identical* patterns to a single model, it preserves the high prediction accuracy, low training and inference times per model as evaluated in Kleio. Instead, we assume perfectly accurate access pattern predictions, via a-priori knowledge of the memory access patterns through the collected traces.

Application Runtime

We first evaluate the application runtime and machine learning overheads, when using Coeus to configure the deployment of a machine intelligent page scheduler, such as Kleio, compared to standalone execution of Kleio. We also include a comparison with the case when Kleio operates at very fine-tuned page scheduling periods (`Kleio + OPT period`), compared to 0.01 seconds that was selected for its original evaluation. We have shown in Section 6.3, how the selection of 0.01 second periods is sub-optimal even for purely history-based page schedulers. This comparison will isolate the performance improvements of Coeus that derive from the page scheduling frequency itself vs. the effects of page clustering. Kleio with optimal period selection manages the same pages as in Kleio, since there is no page clustering involved.

Page clustering effectiveness. Figure 6.10(a) shows how many more pages Coeus manages intelligently via its page clustering mechanism, compared to Kleio as a baseline ($1\times$). On average, Coeus manages almost $3\times$ more pages than Kleio, when training the same number of machine learning models. Coeus works exceptionally well for workloads with complex patterns, such as `quicksilver`, for which it manages almost $7\times$ more pages than Kleio, due to the use of a page scheduling frequency that enables the creation of large page clusters with identical patterns. Similarly, Coeus manages $2\times$ - $3\times$ more pages

for workloads with high regularity in access patterns, such as `backprop`, `kmeans` and `hotspot`, helping observe these patterns at a granularity that facilitates their clustering. Similarly, Coeus works well for certain applications with irregular access patterns, such as `bptree` and `pennant`. These specific workloads have part of their memory footprint accessed randomly and another part accessed fairly regularly. Coeus helps cluster together the latter subset of pages. In contrast, the page clustering of identical patterns is not quite possible for workloads with completely random accesses across all pages, such as `bfs`, decreasing memory footprint, such as `lud` and sparse matrix operations, such as `cpd`.

In conclusion, Coeus accomplishes its **design goal G1**, reducing by almost $3\times$ the machine learning models trained, thus the aggregate overheads and resource requirements for their deployment. There are two reasons for this drastic reduction. First, Coeus's sophisticated selection of page scheduling frequency facilitates the creation of large page clusters to be managed with a single machine learning model. Second, the clustering of identical patterns as input to a single RNN model, does not increase the per model training times. In this way, the reduction in the number of RNN models trained, reduces the aggregate learning overheads.

Breakdown of performance gain. Figure 6.10(b) captures the application performance improvement of Coeus and Kleio with fine-tuned periods (Kleio + OPT period) compared to the original configuration of Kleio ($1\times$). On average, Coeus improves performance by $3\times$. There are two reasons for this significant performance improvement, that relate to Coeus's selection of page scheduling periodicity. The sophisticated period selelction enables more effective page clustering, as well as better tuned operational frequency of the page scheduler itself. The configuration of Kleio at an optimally tuned frequency (Kleio + OPT period) captures the maximum extent of performance increase due to the operational frequency itself. Its difference from Coeus isolates the effect of better page clustering alone, on application performance. Breaking down the performance of Coeus, we observe that, on average, $2\times$ of the performance enhancement comes from the operational frequency itself

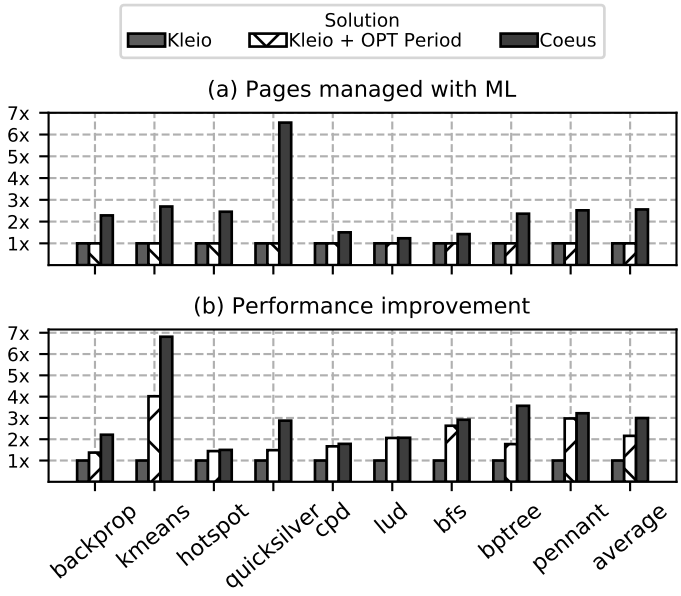


Figure 6.10: Evaluation of standalone Kleio vs. using Coeus, for the *same* number of RNNs, that is the required by Coeus to deliver best performance.

(Kleio + OPT period) and the rest $1\times$ comes from the page clustering.

Regarding the contributions of the page clustering to application performance (difference in the bars of Coeus vs. Kleio + OPT period), we see that in certain cases, such as `kmeans`, it improves performance by $3\times$. Similarly, for `backprop`, `quicksilver` and `bptree` the isolated performance increase that corresponds to the page clustering ranges from $1\times$ - $2\times$. These performance improvements result from using RNNs for more pages, and applying more efficient data tiering and data movement decisions to larger portion of the application working set. However, the management of more pages intelligently, does not always result in application performance improvements. This is the case for applications like `hotspot` and `pennant` whose active memory footprint fits in the available DRAM capacity. Therefore, even though we can make more accurate access pattern predictions with the machine intelligent management of almost $2\times$ more pages, these extra pages do not need to migrate, thus it does not reflect on performance. Finally, for applications such as `cpd`, `lud` and `bfs`, where zero inertia page clustering is not so effective, Coeus still provides performance improvements of up to 10% - 30%.

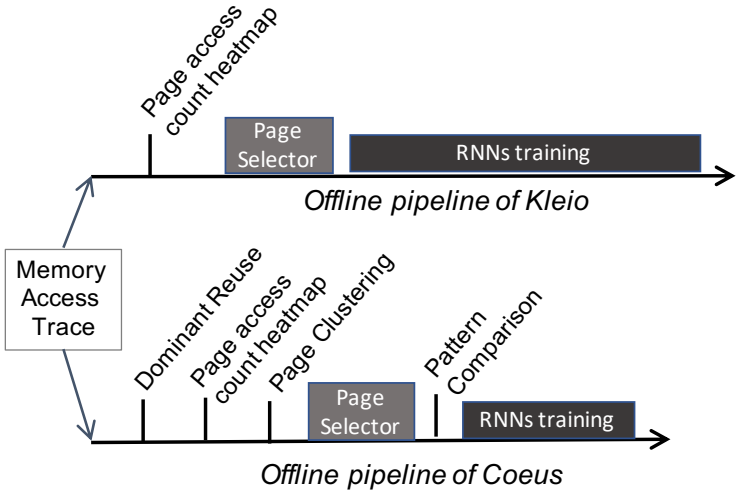


Figure 6.11: Offline overheads of running Kleio vs. Coeus alongside Kleio.

Regarding the contributions of the operational frequency to application performance (Kleio + OPT period), these are significant across all workloads. The fine-tuned page scheduling frequency contributes, on average, $2\times$ of the performance improvements of Coeus, and is significant even in cases where Coeus clustering alone makes small contributions to performance. This reveals an opportunity for Coeus to improve Kleio, by fine-tuning the operational parameters of machine intelligent memory managers, to reduce the management overheads and improve the performance impact.

In conclusion, Coeus accomplishes its **design goal G2** delivering $3\times$ higher application performance from existing machine intelligent page schedulers. There are two reasons for this drastic increase. First, the page clustering and intelligent management of more pages, improves data tiering and data movement selection. Second, the insight-based selection of page scheduling periods improves the operational frequency of the scheduler itself.

Overheads of Using Coeus

Next, we demonstrate the operational overheads of using Coeus prior to Kleio, compared to the standalone execution of Kleio, as shown in Figure 6.11. Prior to workload execution, Kleio takes as input the workload's memory access trace, converts it to a heatmap of

per page access counts across scheduling periods. This operation is trivial and involves a single analytical pass to the memory trace, thus has runtime linear to the total number of memory accesses. Then, Kleio runs its Page Selector to identify which pages need machine intelligent management and deploys a single RNN per selected page.

Coeus enriches Kleio's offline pipeline with few more operations with trivial overhead. First, Coeus extracts data reuse information with a single pass to the memory access trace, to calculate the dominant reuse for setting the duration of the page scheduling periods. Then, similarly to Kleio, Coeus converts the trace to a heatmap of page access counts across periods. Next, Coeus groups pages into clusters with identical page access patterns. This operation leverages data structures like sets, thus incurs runtime linear to the total number of pages, which is even more trivial than the time to create the heatmap. Coeus runs Kleio's Page Selector component 'as-is', thus incurs the same overhead. Coeus then selects which patterns to train RNNs for, with a simple comparison of the page clusters with the pages returned by the Page Selector. Finally, Coeus trains on average $3\times$ fewer RNNs than Kleio, as evaluated previously, thus incurs only a third of the learning overheads.

In conclusion, Coeus comes with minimal operational overheads, realizing its **design goal G3**. It also drastically reduces the learning overheads, overall reducing by almost $3\times$, on average, the offline pipeline of hybrid memory management prior to workload execution.

6.6 Chapter Summary

In this chapter we presented a solution to reduce the learning overheads of machine intelligent page schedulers for hybrid memory systems. In order to be practical and effective, such memory schedulers focus the machine learning on a carefully selected page subset and train per page models in parallel. Yet, the resource and time requirements for training these models vary up to $9\times$ across workload classes and input sizes. To reduce the aggregate learning overheads it is necessary to increase the granularity of the application's memory address space for whose patterns single models are learned. Widely used data clustering

methods, such as k-means, incur non-trivial configuration constraints and overheads, given the high dissimilarity of pages selected for machine learning-based management. In response, we build Coeus, a page grouping mechanism that enables machine intelligent page schedulers to train, in parallel, different models per large page clusters. Coeus is extremely lightweight and highly effective, compared to data clustering methods, because it leverages data reuse insights to fix the granularity of page access patterns, transforming "alike" patterns to completely "like" ones. As a result, Coeus reduces by almost $3\times$ the learning overheads, by managing more application pages under a single model, thus decreasing the aggregate number of models, without increasing the training time of the model itself. Choosing the granularity at which page access patterns are clustered and analyzed, enables Coeus to improve application performance by $3\times$, compared to the performance levels enabled by the configuration of existing machine intelligent page schedulers.

CHAPTER 7

REDUCING OPERATIONAL OVERHEADS WITH PATTERN VISUALIZATION

So far this thesis built mechanisms to maximize the performance and efficiency of hybrid memory management by leveraging insights on data reuse distance, page-level access behaviors and pattern similarity across neighboring pages. These insights were primarily derived by visualizing memory access behaviors and relations. Images can capture copious information that is much more cumbersome to extract through analysis, thus have the potential to reduce system operational overheads. This chapter aims to explore the feasibility of integrating computer vision methods over image-based mechanisms in the context of hybrid memory management. As a use case, we explore the effectiveness of an image-based approach for selecting pages for machine learning-based management, by building **Cronus**[1]. This chapter reveals spatial and temporal correlations across pages, through appropriate visualization of the memory access patterns. Kleio is not able to capture relations across pages, since it operates at the granularity of a page and learns per page access patterns. We explore the integration of visualization in hybrid memory management and propose an image-based page selection process, that drastically reduces Kleio's operational costs, while preserving the effectiveness of the page selection and achieved performance levels.

7.1 Overview

In Chapter 4 we described Kleio, a machine intelligent page scheduler for hybrid memories. Kleio aims to improve application performance levels with machine learning-based memory management, in return for reduced overheads associated with training and deploy-

[1]The name is inspired by ancient Greek mythology, where Cronus (Kronos) was the King of the Titans and the god of time.

ing the ML models. Thus, Kleio focuses on selecting a small page subset whose ML-based management will boost application performance. The process of identifying such pages is not trivial. Kleio deploys performance estimate models, customized for the configuration of the hybrid memory platform, and executes those models repeatedly to produce runtime estimate curves. Since Kleio's design is centered around performance and focuses its operation on a per page basis, it doesn't capture temporal and spatial access characteristics across pages.

Problem Statement. Our current approach in practically integrating machine learning in hybrid memory management includes performance-based methods to select subset of the pages for machine learning model training. Focusing on maximizing application performance for the given configuration of the hybrid memory platform, our performance-based page selection method incurs non trivial execution overheads. In this part of the thesis, we explore the use of computer vision techniques, such as visualization and pattern recognition, for the purpose of building a lightweight mechanism to select the page subset for machine learning-based page scheduling over hybrid memory systems.

The specific contributions are:

- We visualize the pages selected for machine learning-based management by Kleio and reveal spatial and temporal correlations (Section Section 7.2).
- We propose the idea of integrating image-based analysis as part of the page selection process. We explore factors such as the image size, resolution, mapping pixels to page access information. We leverage computer vision methods to detect and extract areas of the images that include page access patterns. We couple all these into the design of Cronus (Section 7.3).
- We evaluate Cronus against Kleio, with respect to the page selection overheads and quality of the selections as they reflect on application performance through Kleio's ML-based management (Section 7.4).

472, 467, 463, 471, 613, 464, 468, 593, 597, 601, 605, 476, 475, 589,
466, 604, 488, 600, 484, 609, 483, 487, 461, 465, 469, 473, 612, 608,
470, 596, 462, 479, 381, 616, 523, 592, 474, 480, 486, 477, 481, 485,
595, 482, 614, 478, 611, 615, 603, 599, 602, 610, 598, 606, 607, 594,
591, 590, 655, 380, 387, 516, 517, 379, 769, 315, 489, 377, 522, ...

Figure 7.1: Page IDs of `cpd` selected for machine learning by Kleio.

7.2 Visualization Insight

In an effort to reduce the page selection overheads, we first aim to better understand the relations (spatial and temporal) of the pages selected for machine learning-based management by Kleio, as summarized in Section 4.5. Figure 7.1 shows, for the `cpd` workload, how pages are prioritized for machine learning according to their benefit factor that combines page hotness and misplacements by history-based page scheduling. We observe that pages are not consecutive, not even in a specific range of spatially neighboring pages. Thus, the exact page priority ordering is not sufficient to reveal any insights about the characteristics of the selected pages.

Throughout this thesis the most successful way to derive insights was through visualizing certain behaviors. For example, for Cori visualizing how the page scheduling periods 'break' the page access patterns (Figure 5.2) and capturing the effect on performance, led to the insight to 'not break the data reuse' (Section 5.3). For Coeus visualizing the page-level access patterns for fine- vs. coarse-grained page scheduling frequency (Figure 6.4), led to the insight of how similar patterns transform to identical ones (Section 6.3). Clearly, *a picture is worth a thousand words* and visualization is a great way to reveal insights and build robust solutions.

Therefore, we visualize the memory access patterns of the workloads in Table 3.1. We plot the memory accesses as scattered points which correspond to the sequence of memory requests (x-axis) to virtual page identifiers (y-axis). Figure 7.2 shows this visual representation together with a coloring scheme that captures the page priority ordering.

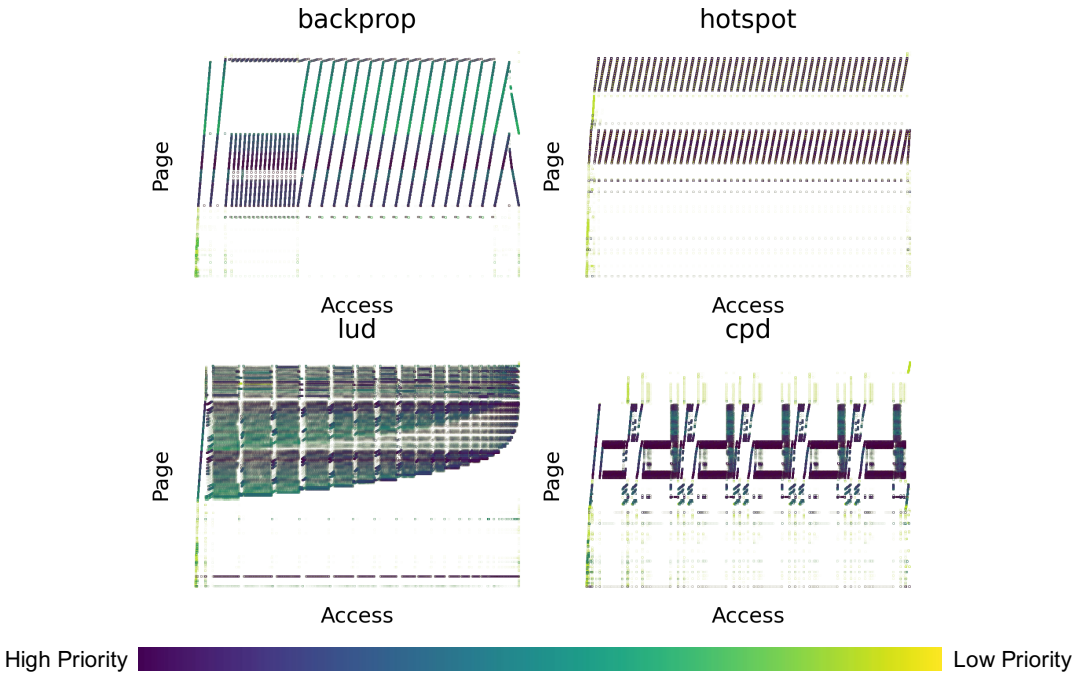


Figure 7.2: Visual representation of a workload's page access sequence, colored according to the page priority ordering for machine learning-based management.

Points with darker color correspond to pages with higher priority, and lighter color to those with lower priority, as depicted in the horizontal colormap bar.

The images clearly reveal spatial and temporal correlations across the pages prioritized for machine learning, that are not obvious when looking at the specific priority order, as in Figure 7.1. *Spatially* neighboring pages that are part of distinct *temporal* access patterns receive similar priority. This is due to the fact, that these groups of pages receive similar levels and patterns of page hotness across runtime, thus are managed very similarly when using history-based predictions. The misplacement metric, that Kleio considers, is particularly visible in the case of `cpd`, where the page group at the top of the image receives low priority even though it has relatively high page hotness. For the specific experimental setup and hybrid memory configuration, that is described in Chapter 3, this page group ends up benefiting from history-based page scheduling.

Takeaways. Visualizing the pages selected for machine learning-based management at the granularity of the workload's memory access patterns through time and space, reveals

new insights on the characteristics that lead to their selection. Pages that are neighboring in space and part of distinct access patterns in time receive similar levels of priority for machine learning, since they have similar access behavior *across* the workload's runtime.

7.3 System Design of Cronus

In this section we describe the system components of Cronus, an image-based pipeline for selecting the pages for machine learning-based hybrid memory management. We discuss the challenges of building such a system and the insights that lead into the system design choices. Figure 7.3 depicts the internal functionality of Cronus and how it interacts with ML-based hybrid memory managers, summarized in the following steps:

1. **Image Creation.** This step visualizes the raw data of a memory access trace, mapping pages (y-axis) and requests (x-axis) into pixels of an image.

2. **Pattern Detection.** This step identifies parts of the image that correspond to groups of pages participating in access patterns across application runtime.

3. **Page Selection.** This step processes the image to extract the range of pages that correspond to the previously detected pattern and prioritizes them for ML-based management given their overall hotness across application runtime. The selected pages are then fed into the ML-based hybrid memory manager for per page training of ML models.

Image Creation

As we have shown earlier in Section 7.2 plotting the virtual page access sequence from a collected memory access trace, visualizes the page access patterns in a way that is comprehensible to the human eye. Thus, we can create a single black-and-white image out of a sequence of page accesses across the application runtime. The choice of black color is

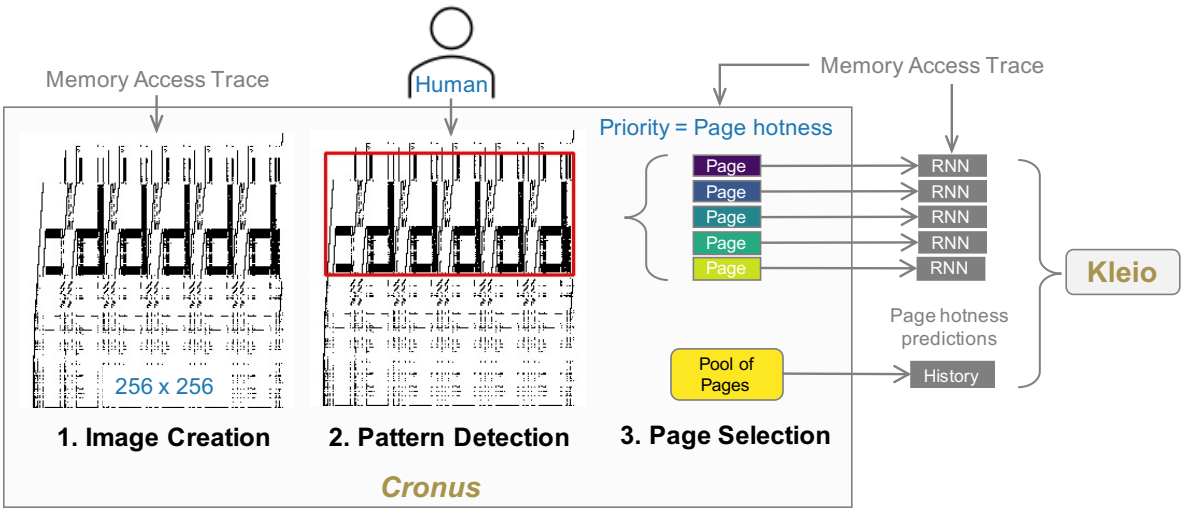
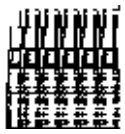


Figure 7.3: System design of Cronus.

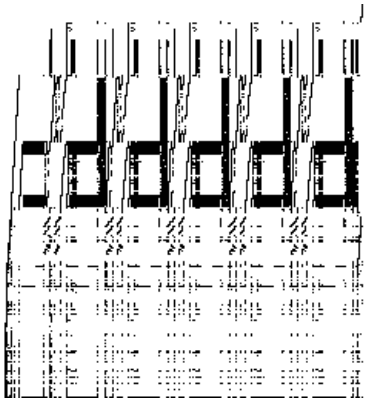
enough to capture the memory access trace information and uses less bytes for storage than a colored RGB image. The resolution of the image is critical because it will determine the extent to which memory access patterns are visibly comprehensible.

Image Resolution. The most intuitive approach is to map a single page access out of the memory access trace to a single pixel of the image. However, this 1-1 mapping will create images of massive size as there can be hundred thousands of pages on the y-axis and accesses on the x-axis. Instead, the average image resolution across datasets used for computer vision and machine learning, such as ImageNet [79], is 256x256 pixels. We explore the creation of images with increasing resolution from 64x64 pixels up to 1024x1024 pixels in exponential size increments, as shown in Figure 7.4 for the `cpd` workload. We observe that small image resolution e.g., 64x64 and 128x128, is not enough to clearly depict the patterns for the given size of the workload. Many pages map to a single image pixel, thus their requests aggregate over few pixels and create very dense areas of black pixels. As a result it is hard to distinguish which pages (y-axis) participate in the sparse tensor operations that the workload performs, as summarized in Table 3.1. In contrast, higher resolution of 512x512 and 1024x1024 enables details of the patterns to be visible, such as the strided accesses over parts of the memory. The middle range of resolution 256x256 is enough to

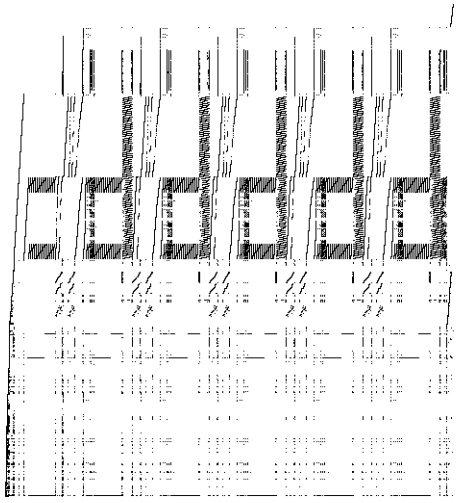


(a) 64 x 64 pixels.

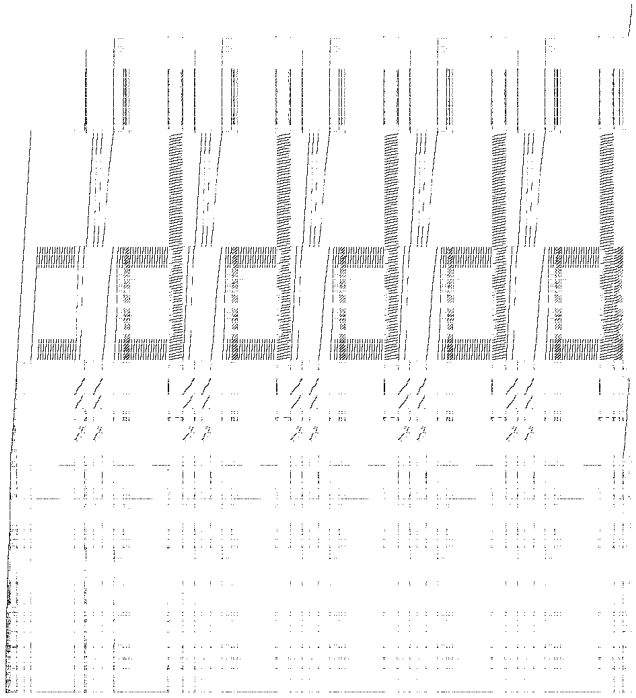
(b) 128 x 128 pixels.



(c) 256 x 256 pixels.



(d) 512 x 512 pixels.



(e) 1024 x 1024 pixels.

Figure 7.4: Effect of the image resolution to distinguish memory access patterns.

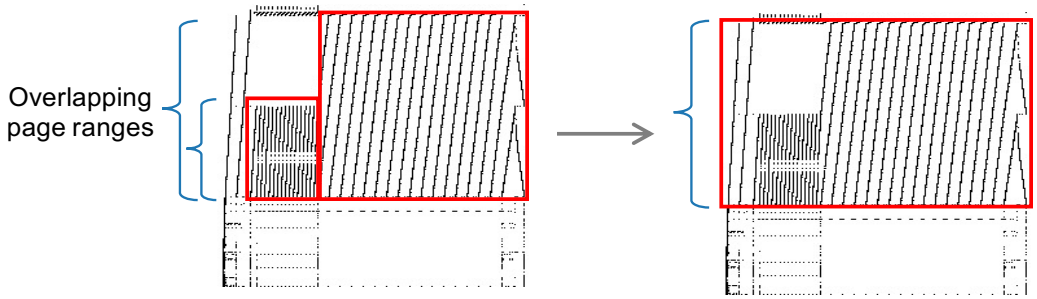
visualize the overall pattern, but not further details.

The decision of which image resolution to use is not trivial. On the one hand, high resolution allows for clear visibility of the details of the type of memory access patterns, such as accesses strided across memory, something that is valuable for the purpose of access pattern classification [80, 81]. On the other hand, high resolution creates images with higher storage and processing time overheads. This is the reason why image-based machine learning uses small resolution, such as less than 256x256 in ImageNet [79]. For the purpose of designing the prototype of Cronus, a system that selects pages for ML-based management, the middle range of image resolution (256x256) is high enough to capture the overall trends in memory access patterns and extract the corresponding pages that participate in the pattern. The details of the patterns will be derived from the raw data of the memory access trace, that Kleio then uses to train per page recurrent neural networks for the selected pages. Thus, Cronus uses 256x256 across workloads for their given sizes. Each pixel will represent $\texttt{pages\_per\_pixel} = \#\texttt{pages}/256$ number of pages and $\texttt{accesses\_per\_pixel} = \#\texttt{accesses}/256$ accesses to these pages.

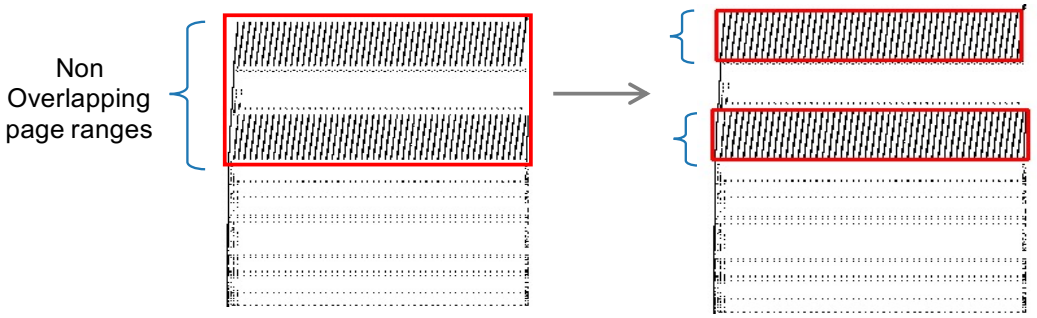
Pattern Detection

After visualizing the page accesses into images, the next step is to detect areas of the images corresponding to patterns across the application runtime. The simplest way to detect such patterns on an image is manually marking areas of the image that correspond to patterns with bounding boxes via human observation. Of course a human-based observation is subjective, thus it may lead to cases of sub-optimal page selection, whose effectiveness we evaluate in Section 7.4. We next describe the methodology we use to determine the positioning of the bounding boxes depending on the workload's access patterns depicted on the generated image.

We consider two cases, when there are patterns across time over page groups that overlap in space and page groups that are distinct. Figure 7.5a shows the memory access patterns of `backprop`. Intuitively, we would mark the two set of strided patterns separately.



(a) `backprop` workload.

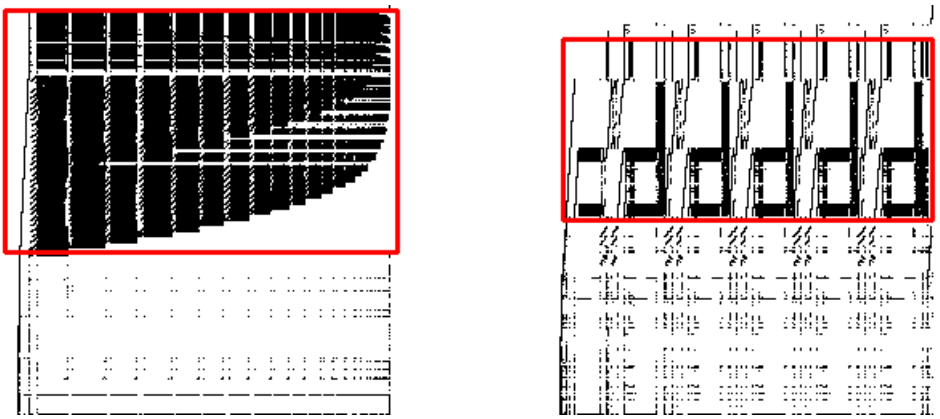


(b) `hotspot` workload.

Figure 7.5: Pattern detection methodology.

However, the corresponding pages (y-axis) overlap in space, thus for the purpose of building an image-based page selection pipeline, we can create a single bounding box that encompasses the full page range. Second, Figure 7.5b shows the patterns of the `hotspot` workload where we can observe two distinct sets of strided patterns. In this case, we mark two separate bounding boxes instead of one because the corresponding page ranges do not overlap. In this way, we avoid including into the selection the horizontal white space betweet the two patterns, that correspond to pages that were only initially accessed and should not be selected for ML-based management. Following the above methodology, Figure 7.6 captures the detected patterns for the `lud` and `cpd` workloads. Regarding the implementation of the pattern detection step, we use functions from the OpenCV Computer Vision Python-based library, to pop-up the image for user interaction and mark regions with the mouse cursor to create the red bounding boxes.

Human-based pattern detection. The pattern detection step can be automated by training ML classifier models over a dataset of images, created with the methodology described in



(a) `lud` (b) `cpd`

Figure 7.6: Page access pattern visualization with 256x256 pixel resolution. The red bounding boxes correspond to the manual pattern detection by the user.

the first step of the Cronus pipeline. Although an automated step is desirable for a system-level component, there is great value behind human-based pattern detection. For the ML classifier to be trained, the image dataset needs to be first annotated. The images need to be associated with labels of the objects they contain, and these objects need to be marked. Thus, the step of using human observation to label memory access patterns as objects of the image is necessary to automate the pattern detection process. We share more thoughts on this idea in Section 9.3, as future directions of this thesis.

Page Selection

After detecting the page access patterns and selecting the corresponding image regions, thus pages, the last step is to extract the corresponding pages and create a priority ordering of the pages to feed into Kleio's page-level management. We perform a reverse mapping of the pixels selected to the underlying page identifiers. The resolution of the image will determine the effectiveness of this mapping, as evaluated in Section 7.4. After extracting the range of selected pages, we go back to the raw memory access trace data and calculate the overall hotness of the pages. Then, we order the pages in descending hotness. Different from Kleio we will make no use of performance estimates to test the effectiveness of a purely image-based page selection process, which we further evaluate in Section 7.4.

7.4 Evaluation

In this section we evaluate the effectiveness of the page selection process that Cronus proposes compared to the one designed for Kleio, the ML-based hybrid memory management system that this thesis contributes, as described in Chapter 4. We compare the absolute page priority and selection that the two systems generate, the resulting application performance via the ML-based management, the time it takes to select the pages and the sensitivity of Cronus to the resolution of the generated image per workload.

Page Selection

Figure 7.7 shows the priority ordering of the pages (y-axis) that Cronus generates based on image-based analysis compared to Kleio's performance-based selection. Overall, the orderings between the two solutions are similar across applications. Since the visualization approach selects page regions in bounding boxes, it completely ignores (yellow) pages with low priority (light green) outside the regions. There are cases where Kleio will give low priority to pages that Cronus will give a high one, such as for `hotspot` and the pages in the top for `cpd`. This is due to the page misplacement metric that Kleio uses on top of page hotness for its performance-based selection, which Cronus is not aware of since it performs a purely image-based selection. Next, we describe the effect in application performance due to the difference in the number and order of the selected pages.

Application Performance

Figure 7.8 shows the application performance improvements (y-axis) when managing pages with ML following the priority ordering that Cronus vs. Kleio calculate via image-based and performance-based analysis, respectively. The baseline case where zero pages are managed with ML, corresponds to having a purely history-based page scheduling approach, as described in Chapter 3. We observe that the page priority ordering that Cronus calculates, results in application performance that is not as a high as Kleio, yet not significantly lower

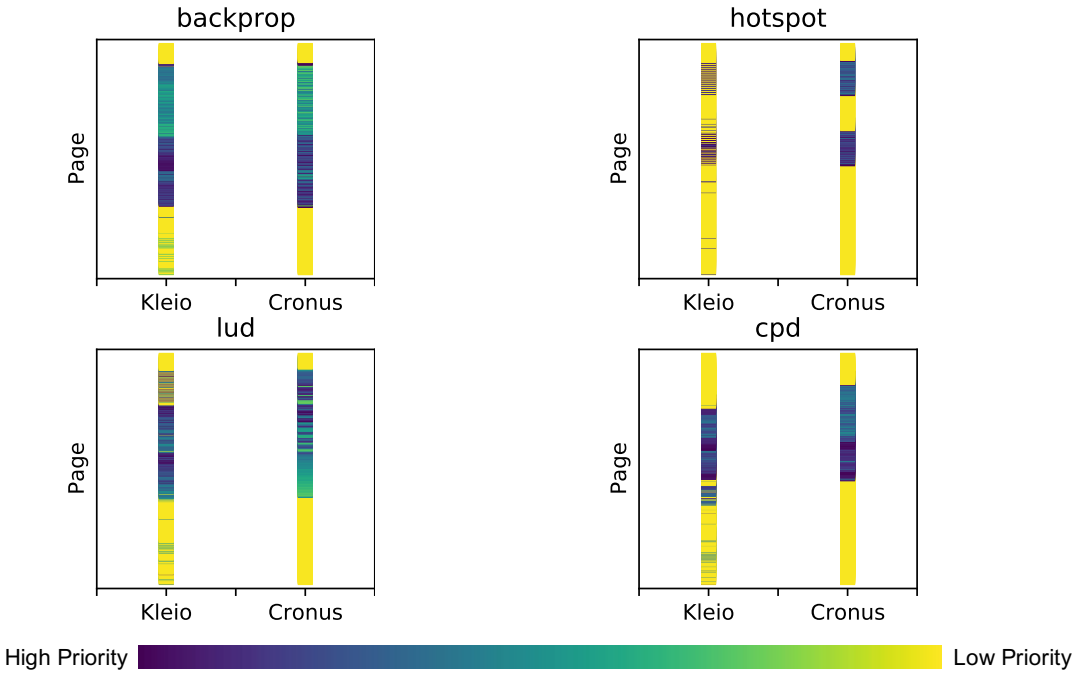


Figure 7.7: Page priority ordering for machine learning of Cronus vs. Kleio.

either. This is to be expected because Kleio as a solution is designed to optimize upon performance, compared to Cronus that is purely image-based. For `backprop` the performance curves of Kleio and Cronus are very close, since the corresponding page ordering shown in Figure 7.7 is very similar. For `lud` and `cpd` performance differs less than 10% across the curve, since the page priorities also have more distinct differences. There are cases, as for the `hotspot` workload, where performance does not improve drastically, due to the fact that there is enough DRAM capacity to accommodate the active working set of the workload, for the configuration described in Chapter 3.

In conclusion, we show that an image-based page selection pipeline (Cronus) can still deliver high levels of application performance, with less than 10% difference across workloads compared to an analytical selection that uses performance estimates. The effectiveness derives from the insights we described in Section 7.2 regarding the spatial and temporal correlations of pages selected for ML-based management by Kleio, that Cronus captures with image-based operations. Cronus design does not consider performance related metrics, such as page misplacements, as Kleio does, yet there is no restriction in future

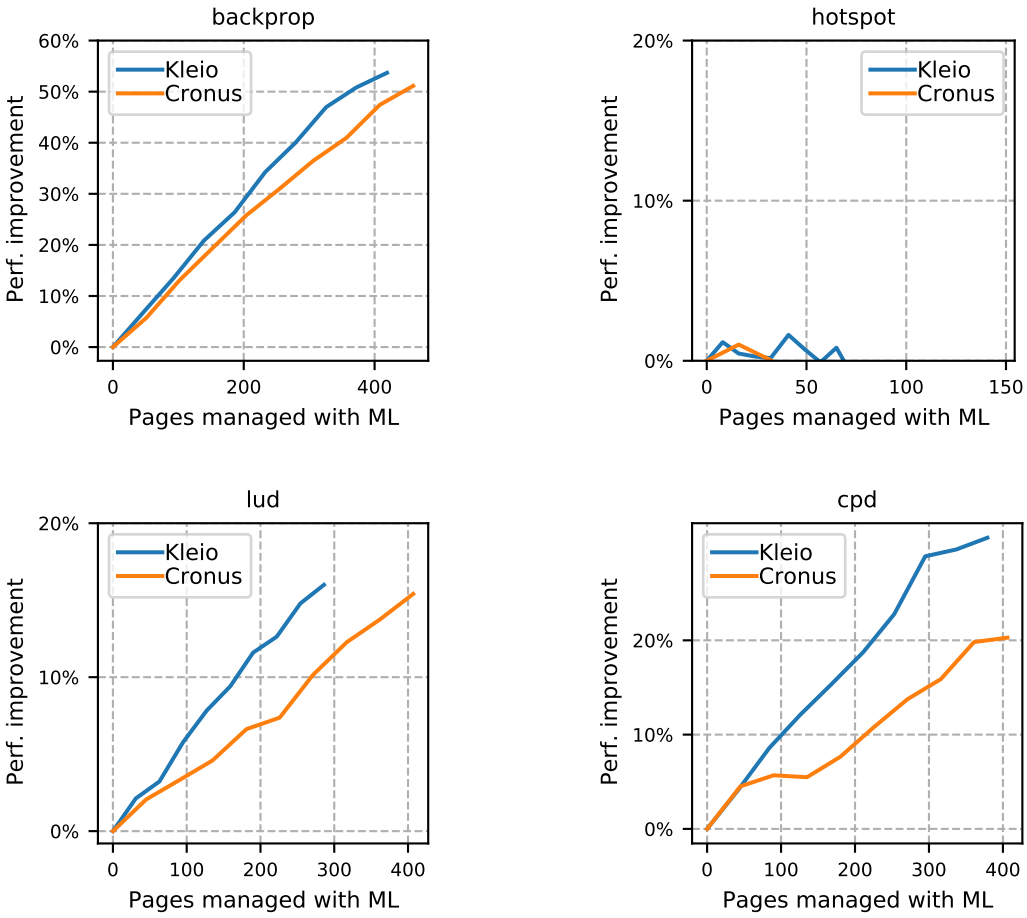


Figure 7.8: Application performance via the ML-based management of the pages in the priority calculated by Cronus vs. Kleio.

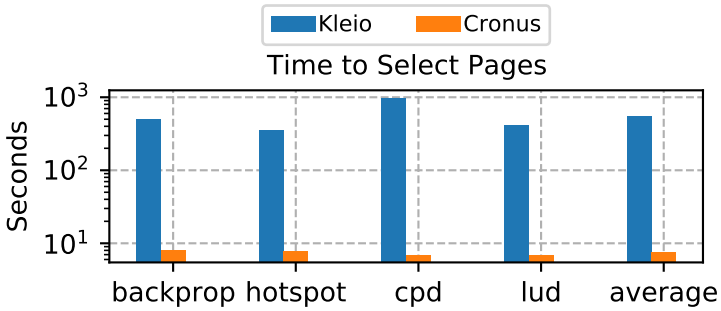


Figure 7.9: Time to select pages between Cronus and Kleio.

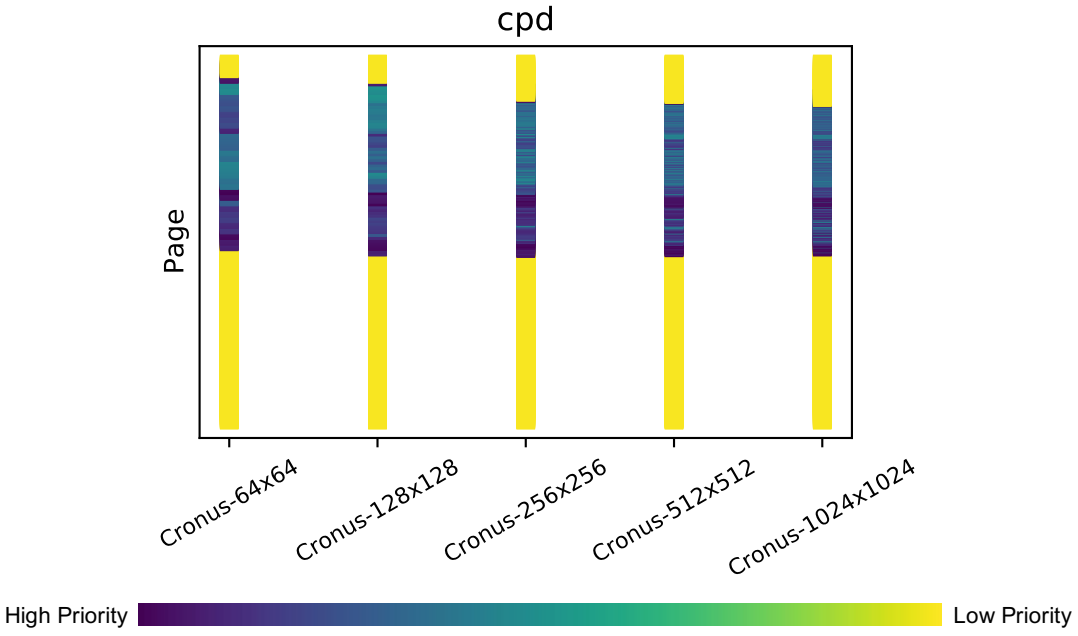
integration of such metrics, as we discuss later in this section. Cronus focuses on capturing the effectiveness of a purely image-based solution, and proves that it can deliver the desired performance levels.

Selection Runtime

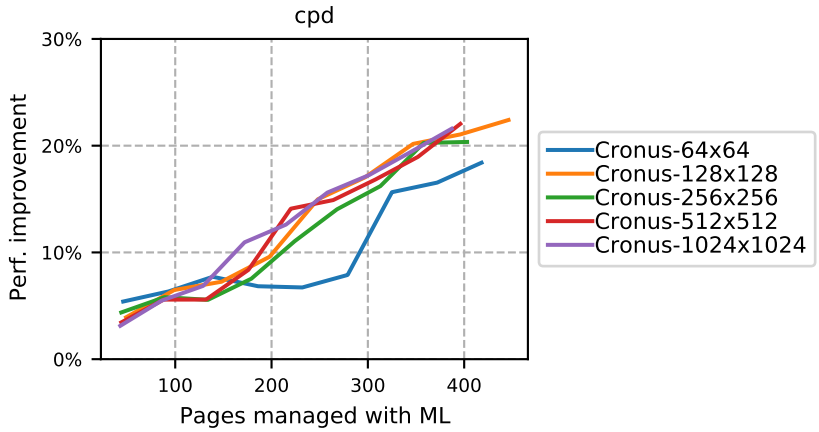
Figure 7.9 captures the runtime in seconds of running Cronus and Kleio's Page Selector component. We observe that Kleio has significant operational overheads, since it involves the repeated execution of application performance estimate models, as we have described in detail in Section 4.5. In contrast, Cronus runs for all applications in less than 10 seconds. This includes the time it takes to analyze the memory access trace, the user to interact with the image and detect patterns, and the time to process the image into the page priority output. Overall, the visualization pipeline reduces the page selection time overheads of Kleio by $75\times$, on average.

Sensitivity to Image Resolution

Figure 7.10 captures the sensitivity of Cronus to the resolution of the images it generates. We focus on the `cpd` workload whose image representations across different resolutions we have shown in Figure 7.4. In this experiment, we assume that in the pattern detection step the user selects the same bounding box to mark the underlying pattern of sparce tensor operations. The resolution impacts the mapping of pixels to pages, thus influences the page



(a) Page priority ordering for machine learning.



(b) Application performance via the ML-based management of the pages in the above priority ordering.

Figure 7.10: Sensitivity of Cronus to the image resolution.

selection step. Figure 7.10a shows the page ordering that Cronus generates over images of `cpd` with increasing resolution. Although the user aims to select the same region of the image at each resolution, the absolute number of selected pages becomes less when the resolution is higher, thus the selection more precise.

The effect on application performance of the image resolution is depicted in Figure 7.10b. We observe that very low resolution, e.g., 64x64, results in slower increase in performance than others. The rest of the resolutions deliver similar performance levels, since the size of the workload is such that there is not much information lost via the page-to-pixel mapping. Therefore, the choice of 256x256 pixels that Cronus makes for the specific set workloads is robust against any sensitivity to the image resolution.

Takeaways

In conclusion, the evaluation of Cronus shows that purely image-based solutions that are designed based on insights, can be effective and deliver levels of performance comparable to solutions that use analytical models based on performance estimates, as Kleio's page selector does. Most importantly, they drastically reduce the associated operational overheads and accelerate the time it takes to complete the selection process, by 75x on average.

7.5 Discussion

Next, we discuss some aspects of the design of Cronus that can further enhance its effectiveness.

- **Image resolution and hierarchy.** The algorithm that our Python-based implementation uses to change the resolution of the image, results in resizing the image, as shown in Figure 7.4. To reduce the resolution, the default algorithm essentially down samples input data by visualizing some and excluding others. Similarly, increased resolution results in larger images through the visualization of more input data into the 2-D image representation. For the purpose of page selection, there is no need

for high resolution, as we argued in Section 7.3. However, for the purpose of pattern classification and detection, there are various re-sampling algorithms that can provide high resolution and detail for a certain image size, by interpolating values across data samples [82].

However, these algorithms have higher computation cost, thus will end up increasing the image rendering times. Enabling some level of resolution hierarchy can be used to amortize the processing times. One approach is using lower resolution for the whole image and marking regions to 'zoom in' and enlarge them with higher resolution, in a separate image, creating a hierarchy. For example, looking at the image of the `cpd` workload in Figure 7.4, we could use 256x256 for the full image and create a separate image with higher resolution that zooms into the pattern that capture the sparse tensor traversal. There is great potential in exploring the trade-offs among the number of created images, their size, the depth of the hierarchy and the associated image processing times and storage overheads.

- **Image metadata.** One other aspect that is worth exploring is having any metadata associated with the created images, especially when thinking about the general applicability of image-based resource management. For instance, the information that can be associated with the images is the number of pages and requests, the size of the pages, the level of data storage hierarchy that the patterns come from (e.g., last level cache misses), any other particular information on the hardware platform and process of collecting the raw data that are visualized, the workload's name, domain and execution phase. This can help in the image annotation and facilitate further analysis based on the images.

- **Automatic pattern detection.** The current prototype of Cronus includes manual pattern detection, a step that is necessary to create ML-based image classifiers that can be trained to automatically recognize similar patterns. However, there can be analyt-

ical ways to automate the pattern detection process. For example, application-level information or compiler-level methods to associate memory regions with data objects / arrays are ways to identify regions of pages that will share access patterns across time. In addition, one could use the methodology of data prefetching techniques that capture differences in the memory addresses accessed across time to capture sequential, strided and other classes of patterns [80, 81]. There is great value in exploring which option is more robust for pattern recognition and has the least operational overheads and dependencies.

- **Reducing the page selection overheads.** There can be other ways to reduce Kleio's page selection overheads. For example, by sampling the memory access trace or running the performance models fewer times, can produce in less accurate version of the performance estimate curve, that Kleio generates. These approaches will most probably reach performance levels in between the one shown in Figure 7.8, compared to an image-based approach (Cronus) and a performance-based one (Kleio). However, the goal of Cronus is to evaluate the effectiveness of a purely image-based approach, focusing on the page selection as a use case, to lay the foundations for future use of computer vision methods in system-level resource management.

- **Enriching the image-based page selection.** Cronus can reach higher performance levels by associating the pages selected from the image with information on the hybrid memory platform where the application will execute. For example, information on the capacity ratios, initial allocation policies, page migration order of execution can help estimate some data movement actions through appropriate modeling and can capture the page misplacement metric that Kleio uses to select pages along with the overall page hotness. In this way, one can improve the quality of the page selection process and the performance levels achieved, in return for increased runtime overheads compared to a purely image-based approach.

7.6 Chapter Summary

This chapter explores the effectiveness of using computer vision methods and image-based decisions in the context of machine learning-based hybrid memory management, that this thesis contributes. Leveraging the power of images to reveal insights and accelerate decisions, we revisit the page selection process for machine intelligent management, proposed in Kleio's initial design (Section 4.5), aiming to drastically reduce the associated operational overheads. This chapter reveals insights on how the pages selected for machine learning by Kleio correlate in space and time by visualizing the memory access patterns. We build a system solution, Cronus, that is a lightweight image-based solution to detect page-level patterns for machine learning. The quality of the selected pages and the achieved performance levels are comparable to Kleio's, while reducing by $75\times$ the page selection time compared to Kleio. Importantly, Cronus shows the great potential of using purely image-based decisions and lays the design foundations for future use of visualization and computer vision methods in memory management, such as image-based memory access pattern classification, recognition and prediction.

CHAPTER 8

RELATED WORK

This chapter summarizes recent advances in various aspects of hybrid memory management, implemented across the software and hardware stack. Then, we reference systems that use machine learning for the purpose of resource management and highlight approaches to reduce the associated learning overheads.

8.1 Software Solutions

In this Section, we summarize recent work specific to hybrid memory management whose implementation spans across the layers of the software systems stack.

Application-level Solutions. Starting at the top of the software stack, inside applications themselves, related works optimize the algorithmic design to perform more efficiently over the underlying hardware. For instance, the k-means NUMA Optimized Routine (knor) library [4] optimizes k-means for modern NUMA architectures and minimizes synchronization barriers. Similarly, algorithm features, common numerical operations, and algorithm structures can be used to direct data placement for conjugate gradient, fast Fourier transform, and LU decomposition of a matrix [5]. In addition, application-level hints are widely used by related works across the software stack to appropriately guide the focus of the hybrid memory management into user-identified critical data structures and regions. To this extent various solutions propose custom data allocation APIs, that require application source code modifications, to improve initial and dynamic data placement of the marked data regions. [6, 7, 9, 17, 11].

Middleware-level Solutions. Regarding contributions at the user library-level, Memkind [8] is a user extensible heap manager that can be adopted by other middleware solutions to improve performance over heterogeneous memories. Similarly, runtime solutions allow

for easy detection of execution phases to properly align data monitoring and movement time intervals. More specifically, Unimem [9] leverages the MPI communication phases to launch data movements and Tahoe [10] aligns task-based execution with corresponding data migrations. Moreover, various middleware-level solutions rely on application profiling of data access behaviors, build performance and data movement cost models to optimize data tiering [6, 7, 11]. Finally, compiler analyses and code generation methods have been proposed to dynamically migrate pages and improve data locality [83].

Operating System-level Solutions. Lastly, operating system-level solutions rely on page access information available on the kernel's page tables, to identify frequently accessed pages and to periodically migrate them. Certain solutions leverage existing NUMA-based page migration support [12, 14, 15, 20, 28], or appropriately extend NUMA-based data balancing policies [16, 53]. Recent analysis of NUMA-based approaches for systems with emerging persistent memory technologies reveals unexpected performance in specific configurations, such as when using huge pages [84]. Spanning across the software stack, Memif [17] introduces a user interface, a user space library and kernel space service to accelerate page migrations across hybrid memory.

Static Data Placement. The complete software-level solution space also includes optimizations in the initial static data placement across hybrid memories. To this end, our own prior work CoMerge [85] proposes a memory sharing placement policy that improves the hybrid memory resource efficiency upon shared use. In addition, we have also built Mnemo [55], a memory sizing and data tiering consulting tool, that permits quick exploration of the cost-benefit trade-offs associated with different configurations of the hybrid memory components used by key-value store workloads. Other related works include CHOPT [86], which aims to be an optimal offline algorithm for data placement over multi-tiered heterogeneous systems. Finally, the effectiveness of various static data placement methods has been evaluated against disaggregated memory systems with non volatile memories [87].

8.2 Hardware Solutions

In this Section, we reference hybrid memory management solutions that are implemented as or assisted by custom specialized hardware.

Hardware-assisted Solutions. Software-level solutions create significant associated overheads, which often are impractical for the decision time guarantees of resource management solutions. To this extent, a significant body of the aforementioned software-level solutions propose *specialized hardware* to reduce critical overheads. For instance, hardware-assisted page hotness tracking with custom hardware elements is highly suggested by operating system-level solutions [13, 12, 18, 19, 15, 20]. The availability of such hardware is critical in providing similar performance levels between the different organization modes of tiered memory, that is software-managed flat memory mode and hardware-managed cache memory mode.

Hardware-based Solutions. Purely hardware-level solutions introduce custom counters to monitor data accesses and enable threshold-based data migration triggers [88, 29]. In addition, additional hardware buffers for data copies enable access to data that is under migration, reducing associated stalls and improving application performance [29]. Custom memory controller hardware is also proposed to enable support for page migrations in disaggregated non-volatile memories [18]. In addition, Mempod [19] builds a clustered architecture that enables scalable migration support over multi-level memories, that outperforms prior work which transparently manages hybrid memories configured in a combination of cache and flat organization [89, 90].

8.3 Machine Learning-based Solutions

This thesis is the first to contribute a machine learning-based approach for the purpose of *hybrid* memory management. Among the plethora of machine learning strategies, this thesis investigates the applicability and practicality of using Recurrent Neural Networks

versus Reinforcement Learning. Both methods have been proposed across various aspects of memory and resource management, exposing various practicality limitations, which we summarize in this Section.

Recurrent Neural Networks. The use of Recurrent Neural Networks (RNNs), in particular Long Short Term Memory (LSTM) models, has been shown to be effective for learning memory access patterns for the purpose of data cache prefetching by Hashemi et al. [59], though its integration is not yet practical according to the authors. Section 4.4 and Section 4.5 describe in detail the design differences between this prior work and our proposed use of RNNs. The major difference is the granularity at which they learn memory access patterns and the achieved prediction accuracy levels. We propose to learn page-level (Kleio contribution) and page cluster-level (Coeus contribution) access patterns, compared to learning the sequence of most commonly observed clusters of memory address deltas. Our approach enables the parallel and faster training of smaller RNN models that reach high levels of prediction accuracy, while facilitating the use of lightweight non ML-based predictions for majority of the application pages.

Most recent optimizations enhance the achieved LSTM prediction accuracy via building hierarchical models [91] or compressed models [92, 93, 94], enabling online predictions and creating the foundations for practical machine learing-based data prefetching. In addition, Recurrent Neural Networks have been used to learn and predict the lifetime of memory objects and reduce the memory fragmentation of huge pages on C++ server workloads [95]. Similarly, a hierarchical LSTM-based approach is taken to learn the throughput of a set of instructions, reaching higher prediction accuracy than state-of-the-art hand-written tools currently used in compiler backends and static machine code analyzers [96]. Finally, Recurrent Neural Networks are deployed in cloud environments to learn spatial and temporal patterns that translate to QoS violations for the purpose of performance debugging [97].

Reinforcement Learning. The semantics of Reinforcement Learning allow for its appli-

cability across resource management problems, but as this thesis and others [98] highlight, its practical use is not guaranteed. To this end, Reinforcement Learning has been proposed for the purpose of garbage collection [99] and data prefetching [100]. Also, it has been explored for job scheduling [101] in cluster environments and placement across heterogeneous compute hardware [102]. In addition, the authors of [103] build a deep reinforcement learning (DRL) framework, named DRLPart, for solving the problem of coordinating the partitioning of multiple resources in commodity servers. Also, reinforcement learning has been explored for learning interconnection routing for adaptive network traffic optimization [104]. Finally, ConfuciuX is an autonomous strategy to find optimized hardware resource assignments for DNN Accelerators using Reinforcement Learning [105].

Other Machine Learning Methods. We also summarize few other machine learning methods that have been recently proposed for the purpose of resource and data management. Perceptron-based Prefetch Filtering [106] increases the coverage of cache prefetchers without negatively impacting accuracy. Latent factor collaborative filtering is used to find the configuration of cloud compute and storage resources that provides optimal cost-to-performance trade-offs [45]. Also, the authors of [107] show how replacing the data indexing part of the database management stack with machine learning-based components can deliver significant performance benefits.

8.4 Reducing Machine Learning Overheads

This thesis contributes methodologies to reduce the significant time and resource overheads associated with training Recurrent Neural Networks over the target memory footprints of applications executing over hybrid memory systems. These are to integrate machine learning models with lightweight existing methods and leverage data clustering. In this Section, we summarize other ways to reduce these overheads and accelerate learning across related works and industry trends.

Reducing the Problem Space. First, we summarize the approaches to reduce the learning

overheads that the machine-learning based memory management solutions we described above take to reduce the overheads associated with training Recurrent Neural Networks. First, machine learning-based cache prefetchers [59] treat learning the memory addresses accessed as a classification problem. To reduce the number of unique classes / addresses, Hashemi et al. cluster the memory address space in different groups and deploy a single RNN per cluster. The number of clusters created is empirically chosen to be six, with no further insight or sensitivity analysis to reason about the impact of the number of clusters. In addition, the authors of other machine-learning based prefetchers [92] compress the classification vocabulary of the address space. More specifically, the authors convert the input and output memory addresses into 16-bit binary format, reducing the parameter size, thus improving upon training times without significant loss in prediction accuracy. Finally, another method used to reduce training overheads is sampling, as used by the authors of [95] who are able to produce accurate enough predictions even when applying a high sampling rate to their input data. Yet, the effectiveness of sampling is inherent to the information captured in the data. High sampling over a memory access trace may result in significant distortion of the pattern in memory accesses.

Accelerating Machine Learning. More generically, a plethora of hardware accelerators have been manufactured by industry vendors [108, 109, 110, 111, 112] or proposed by researchers [113, 114, 115, 116] to optimize training and inference times of machine learning methods. Specific to Recurrent Neural Networks, whose use is proposed in this thesis, there are efforts to accelerate their training and inference times across the software and hardware stack. Starting from the algorithmic design of RNNs there is a continuous effort to accelerate their inner units [117, 118, 119]. Then, moving to the library-level, DeepCPU [68] improves the RNN performance on CPUs by an order of magnitude, while cuDNN optimizes execution over GPUs of RNNs [120] and other deep learning methods [121].

8.5 Image-based Solutions

Throughout this thesis we leverage insights initially observed by visualizing behaviors regarding memory access patterns and relations between performance and configuration parameters. The last part of this thesis explicitly targets visualization as a technique that can aid in resource management, and presents image-based processing and computer vision methods to facilitate the detection and extraction of page-level access patterns. Therefore, we summarize related works that also leverage visualization-based insights and solutions that actually integrate visuals and image-based pipelines, often coupled with machine learning, into their design.

Insights from Visualizing Behaviors. There is a plethora of visualization-based tools to monitor and showcase metrics, behaviors and performance counters across computing environments. Just to name a few, there is Amazon CloudWatch [122] and Grafana [123] for cloud based platforms, Nagios [124] and LLView [125] for high performance computing environments and Intel's VTune [126] for native hardware servers. Observing behaviors visually helps system administrators better configure the systems, manage the resources, identify bottlenecks and fine tune the system operation. For example, regarding memory management in particular, the authors of LLAMA [95], that was mentioned earlier in this chapter, visualize the timeseries of server memory usage across time when using huge 2MB vs. 4KB pages, to highlight the huge page fragmentation problem. In another example, Kaleidoscope is a system that introduces VM state coloring to characterize and classify the different types of memory states of virtual machines in a cloud datacenter, and to use this information to enable better cloud micro-elasticity [127]. In conclusion, using visualization and coloring techniques is a robust and intuitive way to gain insights, monitor and classify behaviors to improve the system operation.

Image-based solutions. Yet, to the best of our knowledge the use of visualization inside systems solutions is still very limited, despite the plethora of visualization-based tools.

This is inherent to the fact that visualization often requires human interaction and observation, to determine the corresponding system-level actions. However, a significant body of machine learning models are build for image classification and object detection. Coupling together computer vision methods to process the image, can automate the pipeline of using images inside system-level solutions. The most popular example is the case of autonomous driving and self-driving cars, where a video frame from the car's camera is processed to produce a steering command [128]. In the financial domain, trading companies argue that they are able to better predict stock market values by training classification models over image representations of the time series data, instead of the raw numerical values [129]. In the bioinformatics domain, proteins are encoded into graphic representations, that are then used to train image-based classifiers and neural networks to predict protein functions enabling high throughput real time analysis [130, 131]. In the environmental sciences domain, timeseries data, that capture earthquake related seismic waves, are transformed into visual representations of their frequencies, called spectograms. These spectograms are then used to enable fast analysis for finding similarities across observed seismic waves and detecting new eartquakes with similar behaviors [132]. In conclusion, there is great promise in using visualization across scientific domains and this thesis aims to lay the grounds for such image-based solutions in system-level resource management.

CHAPTER 9
CONCLUSION

This dissertation contributes system-level mechanisms to enrich hybrid memory management with machine intelligence. This chapter begins with concluding remarks and a summary of the thesis contributions. Then, we present the lessons learned throughout the process of developing this thesis, that are valuable in general systems research and in particular when designing solutions augmented with machine learning. Finally, the chapter concludes with future directions and next steps that build upon and extend the contributions of this thesis.

9.1 Summary

Application data sizes are ever exploding, while the data access patterns of emerging application domains become more complex and irregural. Traditional memory hardware technologies fail to scale in the necessary capacities and speeds to accelerate modern analytics. In response, new hardware technologies with diverse characteristics, such as data persistence, are integrated in the memory subsystem to boost application performance and system cost efficiency. Yet, existing system-level resource management policies and configurations are not effective against this new heterogeneity in the memory hardware and application data access behaviors. This thesis identifies a significant gap in performance (Section 2.2) between current solutions and what could be achieved when optimally managing these emerging hybrid memory platforms. To close this gap in performance, this thesis explores the effectiveness and practicality of using machine learning methods in system-level hybrid memory management. Figure 9.1 gives a visual representation of the system-level components that thesis contributes and how they fit into the hybrid memory management software stack.

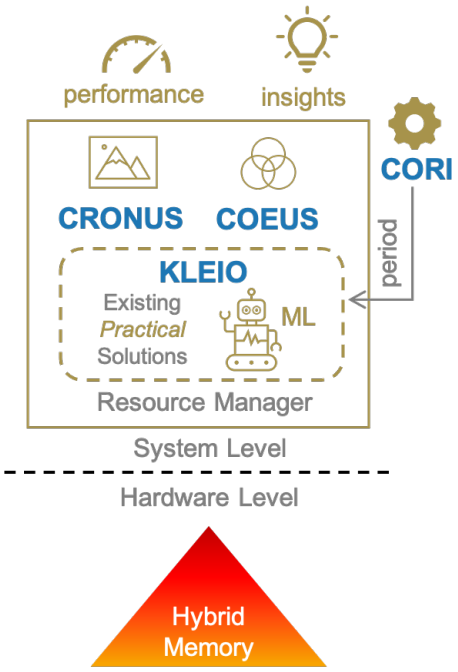


Figure 9.1: Summary of thesis contributions.

Chapter 4 presents the first contribution, Kleio, which makes the case that completely replacing the hybrid memory manager with a machine intelligent component, such as a reinforcement learning agent, is not scalable and robust to hardware configuration changes. Given the massive memory footprints of applications executing over hybrid memories, Kleio identifies a small page subset, whose machine intelligent management boosts application performance. Then, Kleio deploys Recurrent Neural Networks to learn page-level access patterns, while using lightweight existing history-based predictions for majority of the application pages. In this way, Kleio bridges on average 80% of the relative performance gap between existing and oracular solutions, while laying the grounds for practical machine intelligent data management with manageable learning overheads.

The next contribution of this dissertation aims to further boost the performance and efficiency of system-level hybrid memory managers. Chapter 5 reveals that related works do not properly configure their operational frequency relying on empirical tuning, thus failing to deliver up to 100% of performance improvements. The second thesis contribution, Cori, leverages insights on application data reuse to tune the duration of the periodic time inter-

vals at which hybrid memory managers operate. Cori is lightweight and practical, reducing by $5\times$ the number of tuning trials compared to existing empirical or insight-less tuning approaches. In this way, Cori delivers application performance levels only 3% away from the ones feasible via optimal frequency tuning. Such improvements are complementary to the use of machine learning and further boost its effect on application performance.

The third contribution aims to scale the operation of machine learning-based hybrid memory management to cover a larger part of the application memory footprint, while reducing the associated learning overheads. Chapter 6 presents Coeus, a system-level mechanism that groups together pages that share the same access behavior, enabling the training of a single Recurrent Neural Network per page cluster. Coeus leverages the data reuse insights revealed by Cori to fine-tune the granularity at which patterns are interpreted by the page scheduler, increasing the pattern similarity across pages. As a result, Coeus reduces by almost $3\times$ the associated learning overheads compared to Kleio. In addition, Coeus achieves $3\times$ higher application performance, by the combined effects of applying machine learning to more pages and by performing management operations at a fine-tuned granularity.

Finally, the last contribution of this thesis revisits the page selection process for machine intelligent management, proposed in Kleio's initial design (Section 4.5), aiming to drastically reduce the associated operational overheads. Chapter 7 shows how the pages selected for machine learning by Kleio correlate in space and time by visualizing the memory access patterns. This thesis proposes Cronus, a lightweight image-based solution to detect page-level patterns for machine learning. The quality of the selected pages is comparable to Kleio's and delivers similar levels of application performance, while reducing by $75\times$ the page selection time compared to Kleio. Cronus lays the foundations for future use of visualization and computer vision methods in memory management, such as image-based memory access pattern classification, recognition and prediction.

In conclusion, this thesis sets the design foundations for practical and effective hybrid

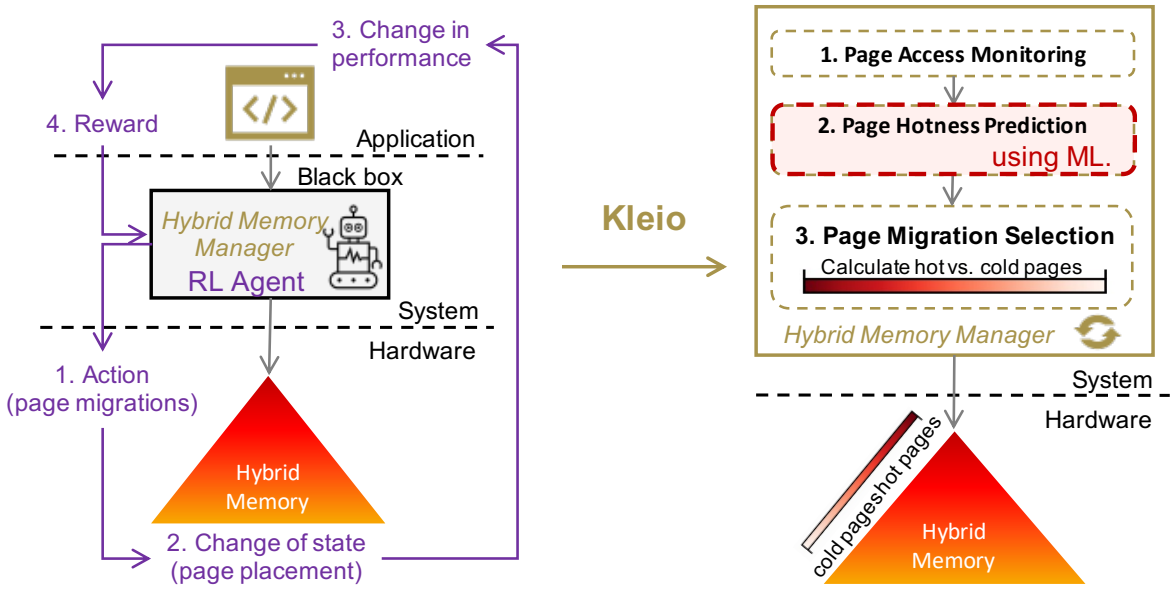


Figure 9.2: Lesson learned 1: Practical machine learning for system-level resource management should learn data access behaviors, not management actions.

memory management enriched with the necessary use of machine learning to boost application performance and system resource efficiency. Most importantly, the proposed solutions are based on insights that allow for lightweight and robust mechanisms that do not add to complexity and overheads of machine intelligent management.

9.2 Lessons Learned

In the process of designing and building the contributions of this thesis, we developed insights and learned valuable lessons that are applicable overall in systems research. Next, we summarize our observations and experiences.

Learn the Behavior, Not the Action

This thesis adds machine intelligence into the system-level resource management of hybrid memories. Thus, the most fundamental design aspect was to decide *which* machine learning method to use and at what part of the systems software stack to integrate it. At first, we explored the applicability of Reinforcement Learning (RL). That meant *replacing* the hybrid memory manager systems component with a reinforcement learning agent,

that learns via taking actions, as depicted in Figure 9.2. The agent would select and move pages across hybrid memory and learn whether this choice of page migrations was beneficial by observing any potential performance improvement due to the new data tiering. However, this type of learning, via observing the effect from the action (data movement), makes the solution dependent on the specific hardware configuration of the hybrid memory environment. If anything changes regarding the number of memory units, their capacity and relative access speeds, in that case different actions may be more beneficial and the RL agent should be re-trained. In addition, the action space in hybrid memory management is exponential, since it involves deciding about the placement of every single page for massive memory footprints. In conclusion, the use of machine learning methods that *replace* systems components in resource management require re-training upon changes in the hardware configuration, breaking the robustness of the system.

To avoid such constraints, we explored the use of machine learning *as part of* the hybrid memory manager's functionality, as depicted in Figure 9.2. A system-level hybrid memory manager periodically monitors page access behavior, projects future access patterns and, according to these predictions, decides which pages to move across hybrid memory following certain heuristics and policies. The most predominant and effective data tiering policy across related works is to move frequently accessed (hot) pages to faster memory technologies. That means that we already have robust policies to decide what *action* to take. The part that needs extra intelligence is making more accurate predictions of future page access patterns given the behaviors observed so far. This is why this thesis proposes the use of Recurrent Neural Networks (RNNs) to learn memory access *behaviors*. In addition, this design choice enables the use of machine learning models for complex behaviors, and simpler and more lightweight prediction models for others. In this way, the integration of machine learning can be practical and effective. In conclusion, the use of machine learning methods that *enrich* current systems components in resource management with the necessary amount of machine intelligence, allow for a practical, effective

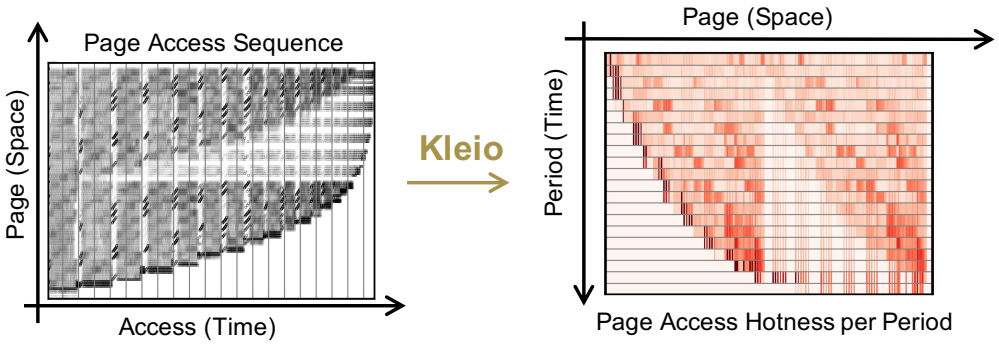


Figure 9.3: Lesson learned 2: Kleio learns memory access patterns at the granularity of a page, to enable the mix of machine learning with lightweight management methods across pages.

and configuration-independent solution.

It is all about the Right Granularity

This thesis reveals three situations where operating at the right granularity is critical for the system's practicality and performance. To begin with, regarding the machine intelligent management of hybrid memories, Kleio trains machine learning models at the granularity of a page. The observations that led to this design choice were the following. At first, the most intuitive way to deploy recurrent neural networks was to use a single model per application, taking as input the overall sequence of page accesses across runtime. In this way the model would learn *which* page would be accessed next. We quickly realized that given the massive memory footprint of the target applications, the model would take days to train and properly tune the hyperparameters. In addition, we observed that the resulting accuracy of the predictions was very low. Suprisingly, related works [59] that have trained single RNN models for the purpose of data prefetching, would consider top-k predictions. For the purpose of hybrid memory management this is not useful, since we wanted an accurate prediction of which pages would be accessed next, so as to move the appropriate pages across hybrid memory.

Therefore, we decided to design Kleio so that it learns patterns at the granularity of a page, not the whole application's memory footprint. Thus, the RNN models would learn

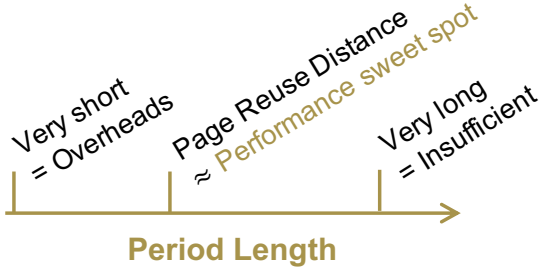


Figure 9.4: Lesson learned 2: Cori tunes the granularity of periodic time intervals when data is moved during hybrid memory management, to maximize application performance and system resource efficiency.

how hot a particular page would be in the future, as depicted in Figure 9.3. This allows for the parallel training of models and faster training since the input for the models is now much smaller in size. Also, we were able to properly tune the hyperparameters of the models and reach high levels of prediction accuracy. Operating at the granularity of a page also enabled Kleio's hybrid approach of managing a small page subset with machine learning and using existing lightweight history-based predictions for majority of the pages. In conclusion, the operational granularity of Kleio allows for practicality and effectiveness.

The second contribution of this thesis, Cori, also highlights the importance of systems operating at a proper granularity to deliver maximum application performance improvements. Cori reveals that setting of operational configuration parameters at arbitrary, insight-less or empirical granularity, may significantly hurt the performance of a system. Related works focus on improving performance through optimizing various design points, such as which pages to move and at what granularity (huge pages vs. regular pages), but ignore the effect of critical parameters such as *when* to move the pages. Cori explores the effect on application performance from the relation of the per page reuse distance in time with the duration of the periodic time intervals when data is moved during hybrid memory management. Cori reveals insights, as depicted in Figure 9.4, and uses them to fine-tune the frequency of data movements by page schedulers over hybrid memories. Most importantly, Cori reveals our poor choice of frequency in our configuration of Kleio. At that time, we

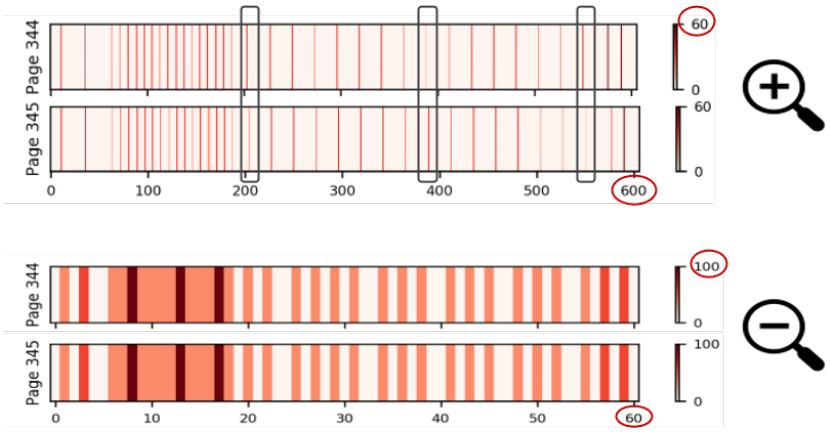


Figure 9.5: Lesson learned 2: Coeus tunes the granularity at which patterns are interpreted by the machine learning-based hybrid memory manager.

did not realize how the extremely fine-grained operational frequency of Kleio allowed data monitoring and movements costs to dominate.

Our third thesis contribution, Coeus, integrates Cori's insights into the operation of machine intelligent page scheduling and delivers significant application performance improvements. In addition, Coeus shows how observing page access behaviors at a more coarse-grain granularity, like zooming out, increases their similarity, blurring out any minor differences. In the context of page access patterns, longer page scheduling periods allow for sequences of page access counts across periods that are completely identical, as depicted in Figure 9.5. Coeus aims to scale the machine learning-based management across more pages, training recurrent neural networks at the granularity of a *page cluster*. Using these observation on page access pattern similarity, Coeus automates the process of page clustering and alleviates the need to employ machine learning data clustering methods.

Keep it Smart but Simple

This thesis explores the use of machine learning in hybrid memory management, aiming to deliver a practical solution, which is not a trivial task. Kleio, goes through an intricate process to identify which part of the application's memory footprint benefits from machine learning-based management. Even though the selected page subset is small compared to

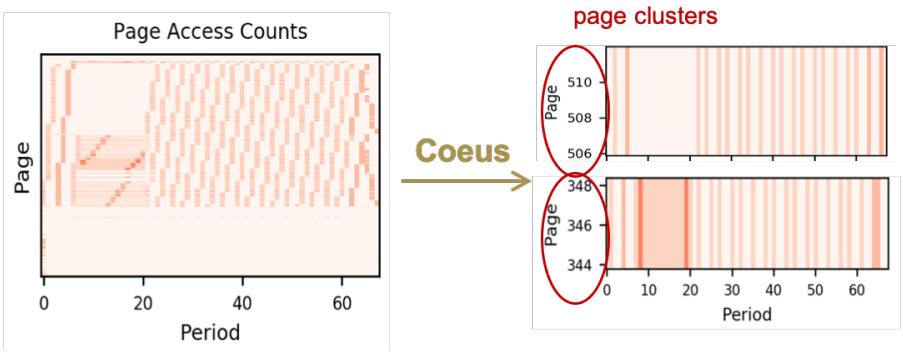


Figure 9.6: Lesson learned 3: Coeus clusters together pages that share the same page-level access patterns, bypassing the complexity of configuring and overheads from integrating unsupervised learning data clustering methods.

the application data scale, the learning overheads are still non negligible. This thesis contributes solutions like Cori, Coeus and Cronus, whose functionality complements and aims to boost the effects of machine learning-based management with Kleio. Therefore, we were very careful to design these solutions to be lightweight and to not further complicate the hybrid memory management process and introduce overheads that hinder their practical use.

Yet, the use of machine learning methods in solving systems problems is particularly popular, and it is interesting to show the extent of their effectiveness. We were also initially excited to see the effects of using unsupervised learning data clustering methods, such as k-means, for the purpose of scaling the granularity of training RNN models per page clusters, that Coeus explores. Quickly we realized that the integration of k-means introduces additional overheads and extra configuration parameters for fine-tuning. Most importantly, we came to the conclusion that it was not even necessary to use machine learning for the purpose of page clustering. Often, observing a problem from a different angle and at the right granularity enables simpler and more robust solutions. As depicted in Figure 9.6, Coeus simply clusters together the pages that share the same sequence of page access counts across periods of time, by leveraging Cori's insights on fine-tuning the period duration. The use of machine learning in system-level solutions calls for practical,

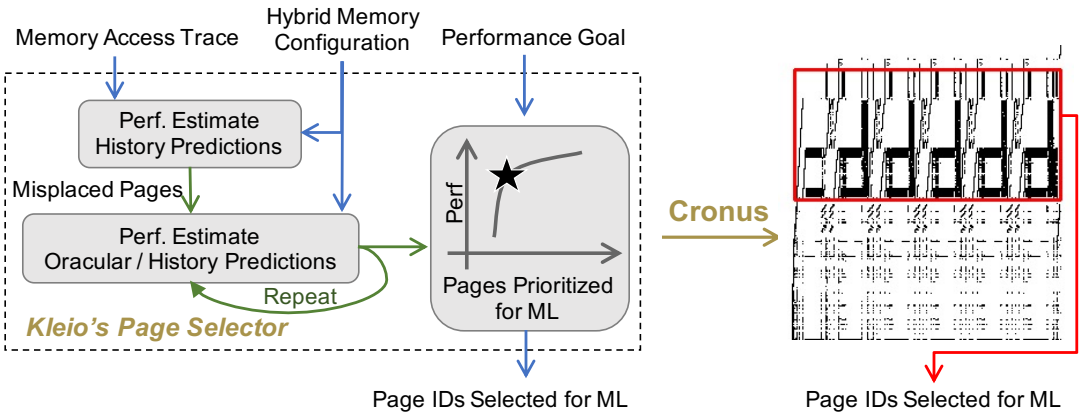


Figure 9.7: Lesson learned 4: Cronus accelerates the time to select pages for machine learning via an image-based approach.

judicious, carefully thought-out methods, applied only when necessary to avoid their non trivial overheads and configuration constraints.

A Picture is Worth 1000 Words

All of the thesis contributions are either enriched or based on insights that derived from visualizing the memory access patterns of applications. In systems research we often try to derive behaviors by observing their effect on performance or capturing meta-metrics, such as page hotness. Instead, this thesis highlights that directly visualizing behaviors is most beneficial to develop insight-based systems techniques. By visualizing the page-level access patterns this thesis produced the following insights.

First, Cori reveals the relation between data reuse and duration of page scheduling periods. This insight originated from visualizing the period lengths as vertical lines on memory access patterns, as shown in Figure 5.2. Observing how the vertical lines 'break' the strided and structured access patterns, while comparing the achieved performance levels, was key to unlock the data reuse insights. Second, Coeus reveals the effect on the patterns of page hotness when varying the duration of the page scheduling periods, thus the relative valua of page acess hotness per period. Visualizing these patterns, as shown in Figure 6.4, across long and short periods unlocked the insight that patterns transition from similar to being

completely identical after certain lengths. Third, Cronus visualizes the page priority that Kleio calculates for machine learning-based management, as shown in Figure 7.2. In this way, it unlocks the insight that the pages selected for machine learning are page clusters that are part of distinct access patterns across runtime.

Finally, this thesis explores the potential of having a purely image-based system component and make use of computer vision methods. The thesis contribution Cronus shows how an image-based approach, as depicted in Figure 9.7, can accelerate the time it takes to select the pages for machine learning-based management by Kleio. Cronus achieves performance levels comparable with the analytical selection that Kleio originally makes and shows great promise for further use of image-based components in system-level resource management. In conclusion, this thesis shows how insightful the use of visualization can be and how there is great benefit in using image-based approaches to speed up system operations. In this way, we believe that *an image is worth 1000 LoC* (lines of code).

9.3 Future Directions

In this section, we propose various directions and ideas that someone can use to build upon, extend and enrich the contributions of this thesis.

Relative Aspects of Hybrid Memory Management

This thesis contributes system-level mechanisms that enable the practical integration of machine learning in hybrid memory management. These contributions focus on improving the selection and fine-tuning the frequency of dynamic page migrations, in a practical and insightful way. Next, we describe some other aspects of hybrid memory management that can further enrich and complement this thesis.

Page size. This thesis proposes the use of machine learning methods to learn memory access patterns at the granularity of a page, assuming a 4 KB page size that is predominantly used across systems. However, there are many benefits from using huge pages, e.g.,

2 MB page size, that emerging memory platforms use [1, 84]. Given the massive memory footprints of emerging applications and workloads, learning patterns at the granularity of a huge page would reduce the associated learning overheads, by reducing the aggregate number of ML models. Essentially, this is what we achieve with the Coeus contribution compared to Kleio's original design. Coeus automatically clusters together pages that share the exact same sequence of page access patterns across periods of time. These are pages that are neighboring in space, as shown in Figure 9.5. In this way, Coeus affects the size of the pages managed with machine intelligence, but keeps the granularity of monitoring and moving pages at 4 KB. Regarding the general management of regular (4 KB) vs. huge pages (2 MB), systems like Ingens [28] and LLAMA [95] resolve fragmentation issues that may arise and systems like Thermostat [12] and Nimble [14] enable transparent huge page management. The operation of these systems is complementary to the systems proposed in this thesis and all together contribute to effective hybrid memory management.

Energy efficiency. In hybrid memory management the dynamic data movements that are triggered to improve application performance through data tiering, directly affect the energy efficiency of the system since moving data requires time and resources. The Cori contribution of this thesis fine-tunes the frequency of page migrations across hybrid memory, by offseting the cost of moving the data with the benefit from achieving data tiering that improves application performance. As a tuning solution, Cori allows the user to select the frequency which satisfies their performance and efficiency levels. Thus, Cori can provide the frequency that maximizes the system's energy efficiency via minimizing the aggregate data movements.

Data object information. Information about application-level data objects / arrays / variables has been used across hybrid memory management solutions to guide data tiering decisions via custom data allocation APIs [6, 7, 9, 17, 11] or compiler-level guidance [83]. These solutions can mark memory regions and keep track of the corresponding application-level data objects. Such information can be beneficial to better guide the page selection for

machine learning-based management. The contributions of this thesis are entirely at the system-level, where there is no information about how pages associate to the application data. As future directions of this thesis, it is interesting to see how application-level data object information can aid in improving the following aspects. First, will machine learning-based management benefit from automatically clustering the pages that belong to the same data object? Can this approach help reduce the ML related overheads, while reaching high prediction accuracy that is comparable to the one provided by Kleio and Coeus? Second, can we use application-level insights to decide which data objects / pages to manage with machine intelligence? Is it the case that application domain experts can have much better understanding of the complexity of the data access patterns, compared to what can be derived at they system-level? How does an application-level page selection compares to the system-level (Kleio) and image-based (Cronus) approach? In conlusion, there is great potential into enriching the contributions of this thesis with application-level information.

Hardware technologies for hybrid memory. The proposed approach of this thesis to apply machine intelligence for learning memory access patterns, instead of hybrid memory management actions (Section 9.2) makes the design robust across configurations of the underlying hardware. Since these are system-level contributions, they can work against any other mix of technologies such as DRAM, persistent memory (PMEM), high bandwidth memory (HBM) and MRAM, when they organized in flat mode, as described in Section 2.1. The systems that this thesis builds are evaluated against hybrid memory platforms with DRAM and PMEM, as is the recently released Intel's Optane persistent memory platform [1] that we use for validation. In the context of this thesis, persistent memory is treated as non volatile memory used to scale the overall systems memory capacity.

Building upon this thesis, it is worth exploring the effect of leveraging persistent memory in machine learning-based hybrid memory management. One use case can be to use persistent memory for storing trained ML models, that can be quickly retrieved for inference or re-training, compared to retrieving them from storage. Similarly, extending the

design to use persistent data structures within machine learning methods. Also, one can explore other aspects of persistent memory such as reliability, write endurance and amplification, in the context of machine learning-based hybrid memory management.

Regarding the future of hybrid memory, disaggregated memory platforms of large scale will prevail in the exascale era of compute with fast speed interconnects, such as CXL, as described in Section 2.1. The contributions of this thesis can be enriched with complementary solutions that provide additional support for data movement over disaggregated platforms [38, 87]. In such platforms the sweet spot of amortizing the data movement cost will shift, thus tuning solutions like Cori are essential. Also, it will be interesting to compare the page subset that Kleio selects to maximize performance via ML-based management in single node vs. disaggregated platforms. Can an image-based page selection process, like Cronus, be effective across both platforms? What additional information on the underlying hardware configuration is necessary to capture?

In conlusion, there is great value into further exploring the effect of hybrid memory hardware technologies and configurations in ML-based management. This thesis contributes a design that is robust and can be further enriched to support operation over disaggregated platforms, and leverage technology characteristics such as data persistence when using PMEM.

Online Adaptive Machine Learning-based Management

The contributions of this thesis lay the grounds for the practical integration of machine intelligence in system-level hybrid memory management. The thesis proposes an insight-based and sophisticated design of how machine learning can be used to enrich existing solutions and maximize application performance and system resource efficiency. We address fundamental questions, such as which machine learning method to use, what exactly to learn and predict, which part of the systems software stack to extend with intelligence? All of that while keeping in mind the exploded data sizes of target applications, that hinder the insight-less use of machine learning over all application data. The thesis builds robust

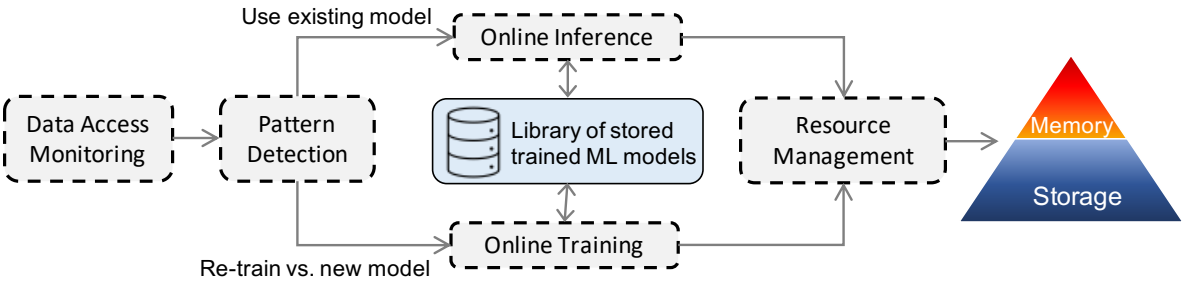


Figure 9.8: Design challenges of online adaptive machine learning-based resource management at the system-level.

foundations for the practical use of machine learning. In addition, the proposed design is not dependent on the underlying hardware configuration, since it uses machine learning to learn the behavior and not the action, as described in Section 9.2. Focusing on exploring the most lightweight design that delivers high levels of prediction accuracy, Kleio performs the page selection, the ML training and inference during an *offline* profiling step, prior to the actual workload execution.

The immediate next steps are to focus on enabling online adaptive training and inference of the machine learning models. Figure 9.8 shows a design draft of such a system that continuously learns. There are many design points that require exploration and insight, as they can introduce interesting trade-offs, which we summarize as follows:

- **Pattern detection.** This will be a critical system component that analyzes the data access behaviors, identifies new or already seen patterns, and makes a decision whether to do inference on a existing pre-trained model, re-train an existing model or build and train a completely new model.

- **Library of trained ML models.** Making the decision between online inference or training requires the bookkeeping of a library of ML models. Where do these models reside? In memory or in storage? Is there a hierarchical storage model?

- **Pattern-to-Model mapping.** There needs to be a way to identify and match the existing models to an observed pattern / behavior. What is the identity function, how

to ensure uniqueness? Kleio simplifies this process and maps a single page to a single model that is identified by the virtual page address. How to extend this mapping to cases where randomization of the address space happens, or the same application executes over different inputs, thus memory footprints?

- **Operation while online training.** While the system does online training and inference, how does it ensure smooth operation and uninterrupted resource management? How frequently is a decision made to do online training, what triggers such a decision? Kleio's hybrid approach enables the use of lightweight history-based predictions as a fallback to the machine learning-based ones, while training is done. Other solutions perform online training periodically [91], training during one period and using the inferred predictions during the next. Is this design robust, or it reveals sensitivity to the quality of the predictions and the accuracy reached?

- **Application runtime.** Long running applications are a better match for online management compared to short ones, since they allow for enough time to do online training and inference during the workload's execution. How can we enrich the system with information on application runtime? Do we make use of explicit information or solutions that leverage machine learning to predict runtimes and resource use [133] and object allocation lifetimes [95]? Or should the resource management system be completely agnostic to application-level characteristics and focuses purely on the underlying access patterns that it observes?

- **Accelerating ML.** Accelerating the runtime of online training and inference are critical in amortizing the associated costs. It is critical to use emerging hardware technologies or other software-based methods, as described in Section 8.4, to enable fast training and inference times, while applications are running. It is also important to ensure that the operation of the resource management system is not affected by changes in the underlying hardware that is used, as this thesis contributes with the

design of Kleio.

In conclusion, this thesis sets the design foundations for online adaptive machine learning-based hybrid memory management. We described our thoughts on how the contributions of this thesis can be extended to online operation and captured the set of challenges that need to be addressed for a robust design.

Coupling Machine Learning-based Management with Computer Vision Methods

The last contribution of the thesis makes a case for leveraging visualization to build image-based components for machine learning-based system-level resource management. Taking it a step further, we aim to explore the use of images for the purpose of memory access pattern classification, recognition and prediction. This ties together with the proliferation of machine learning models trained over image input datasets, such as ImageNet [79] and CIFAR-10 [134]. Across scientific domains there is a plethora of machine learning and computer vision methods coupled together to classify and detect objects, as summarized in Section 8.5.

Similarly, we aim to build a dataset of annotated images, where the objects in the image are categories of memory access patterns across application domains and sizes. The creation of such a dataset requires human based annotation with appropriate labels and metadata depending on the class of the patterns. How can we consistently categorize patterns, so as to enable public contributions from the community? There are various classes proposed across works particularly for the context of data prefetcing [80, 81]. What metadata is necessary to keep to facilitate the mapping of image pixels to memory accesses? Also, the image resolution should be properly set to minimize information loss. Should a hierarchical approach be followed that allows 'zoomin in' to areas of the image? This thesis lays the foundation for the creation of such a dataset, through the suggested visualization methodology and pattern extraction in Chapter 7. Such a dataset will then enable the use of machine learning models to classify and detect and computer vision techniques to ex-

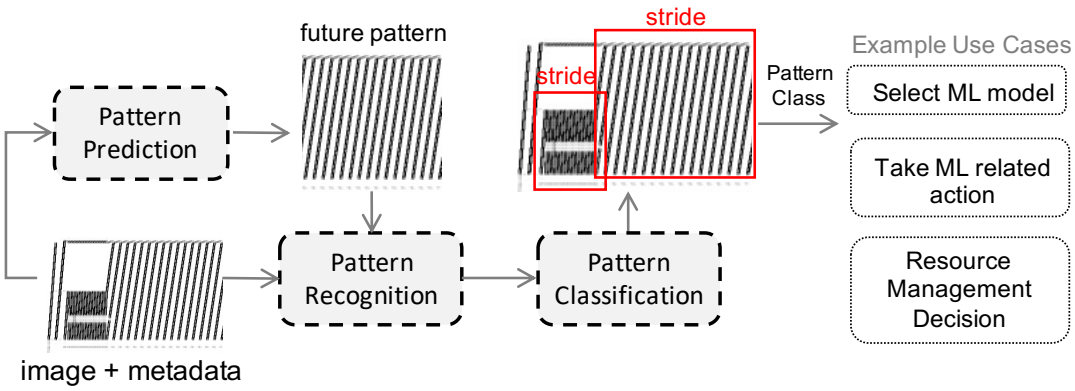


Figure 9.9: Future use of computer vision methods for data access pattern recognition, classification and prediction as part of system-level resource management.

tract such patterns in unseen images. In addition, it will be extremely beneficial to better understand data access behaviors across application domains with contributions from the research community and industry.

Finally, having an image dataset of memory access behaviors can further enrich the machine learning-based system-level resource management with image-based components, as depicted in Figure 9.9. For example, images can now be used to identify patterns, map patterns to machine learning models and trigger any necessary re-training or new model creations, as we described earlier in the section. Moreover, convolutional neural networks coupled together with recurrent neural networks can be used to learn and predict memory access patterns across times based on their image representations. Such machine learning models have been shown to reach higher prediction accuracy than using recurrent neural networks over numerical data [129]. How does an image-based pattern prediction compare against the design proposed in this thesis? What part of the memory footprint corresponds to an image, how many models are trained per application? These are interesting comparison points and there is great potential in improving upon the contributions of this thesis.

In conclusion, this thesis opens up a new research direction that couples machine learning with computer vision for the purpose of resource management. It is exciting to evaluate the extent and the context where such image-based machine learning models will be more

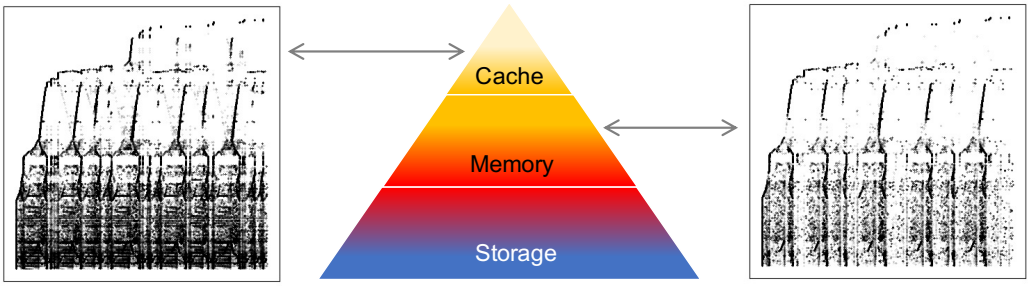


Figure 9.10: Data access patterns change across the data storage hierarchy, as data accesses get filtered across the storage layers.

effective, reach higher levels of performance and reduce learning overheads, compared to training machine learning models over numerical data with features that capture data access behaviors analytically.

Extending the Machine Learning-based Management Across the Data Storage Hierarchy

The contributions of this thesis improve upon current solutions in systems with heterogeneous *memory* hardware by adding machine intelligence. However, there is potential for the proposed approach of integrating machine learning methods to be extended to the management of storage technologies. Our key design point that we propose is to use machine learning to enrich existing lightweight and practical solutions with the necessary machine intelligence and learn behaviors, rather than actions, as described in Section 9.2. This approach can be applied for managing data stored across memory-only hardware, storage-only hardware, across memory and storage, as well as when considering data offloads to GPUs and accelerators [135]. However, the data access patterns will slightly differ across layers, since each layer filters the amount and type of data that reach the lower levels of the storage layers, as depicted in Figure 9.10. Leveraging the similarity of these images presents an opportunity to expand an image-based resource management approach across

emerging computing platforms with extreme heterogeneity [41], such as layers of hierarchical heterogeneous memory and storage, accelerators and accelerator-near memories, and emerging disaggregated platforms.

Final Remarks. This thesis is written at a time where there is a lot of excitement in the research community as well as interest from industry to use machine learning for system-level resource management. We contribute initial steps and design foundations on how to practically use machine learning, while the full integration of machine learning requires a lot more work, as described in the future directions of this thesis. We hope that the lessons learned will inspire and will be of general use to researchers and students that will pursue these topics, particularly since most of the software system solutions that this thesis develops are open sourced for community exploration and contributions.

REFERENCES

[1] *Intel® Optane™ DC Persistent Memory*, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[2] *MemVerge - More Memory. Less Cost.* https://www.memverge.com/more-memory-less-cost/.

[3] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, *Basic performance measurements of the intel optane dc persistent memory module*, 2019. arXiv: 1903.05714 [cs.DC].

[4] D. Mhembere, D. Zheng, C. E. Priebe, J. T. Vogelstein, and R. Burns, "Knor: A numa-optimized in-memory, distributed and semi-external-memory k-means library," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17, Washington, DC, USA: ACM, 2017, pp. 67–78, ISBN: 978-1-4503-4699-3.

[5] P. Wu, D. Li, Z. Chen, J. S. Vetter, and S. Mittal, "Algorithm-directed data placement in explicitly managed non-volatile memory," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16, Kyoto, Japan: ACM, 2016, pp. 141–152, ISBN: 978-1-4503-4314-5.

[6] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, London, United Kingdom: Association for Computing Machinery, 2016, ISBN: 9781450342407.

[7] D. Shen, X. Liu, and F. X. Lin, "Characterizing emerging heterogeneous memory," *SIGPLAN Not.*, vol. 51, no. 11, pp. 13–23, Jun. 2016.

[8] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, "Memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies.," Mar. 2015.

[9] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: ACM, 2017, 58:1–58:14, ISBN: 978-1-4503-5114-0.

[10] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *Proceedings of the*

International Conference for High Performance Computing, Networking, Storage, and Analysis, ser. SC '18, Dallas, Texas: IEEE Press, 2018, 31:1–31:13.

[11] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "Atmem: Adaptive data placement in graph applications on heterogeneous memories," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020, San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 293–304, ISBN: 9781450370479.

[12] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, China: Association for Computing Machinery, 2017, pp. 631–644, ISBN: 9781450344654.

[13] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, vol. 00, Feb. 2015, pp. 126–136.

[14] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 331–345, ISBN: 9781450362405.

[15] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 521–534, ISBN: 9781450348928.

[16] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang, "Hinuma: Numa-aware data placement and migration in hybrid memory systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 367–375.

[17] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 369–383, ISBN: 9781450340915.

[18] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19, Washington,

District of Columbia: Association for Computing Machinery, 2019, pp. 417–427, ISBN: 9781450372060.

[19] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 433–444.

[20] V. Gupta, M. Lee, and K. Schwan, "Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '15, Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 79–92, ISBN: 9781450334501.

[21] *MLPerf - Fair and useful benchmarks for measuring training and inference performance of ML hardware, software, and services.* https://mlperf.org/.

[22] *MLBench: Distributed Machine Learning Benchmark*, https://mlbench.github.io/.

[23] J. Li, Y. Ma, and R. Vuduc, *ParTI! : A parallel tensor infrastructure for multicore cpus and gpus*, Last updated: Jan 2020, Oct. 2018.

[24] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, Boston, MA, USA, 2009.

[25] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Jan. 2011.

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54, ISBN: 978-1-4244-5156-2.

[27] *CORAL-2 Benchmarks*, https://asc.llnl.gov/coral-2-benchmarks/, 2020.

[28] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, Savannah, GA, USA: USENIX Association, 2016, pp. 705–721, ISBN: 9781931971331.

[29] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-based hybrid memory management," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 152–165.

[30] *Pin - A Dynamic Binary Instrumentation Tool*, https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

[31] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19, Phoenix, AZ, USA: ACM, 2019, pp. 37–48, ISBN: 978-1-4503-6670-0.

[32] T. D. Doudali, D. Zahka, and A. Gavrilovska, "Cori: Dancing to the right beat of periodic data movements over hybrid memory systems," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[33] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules," in *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*, ACM, 2019, pp. 288–303.

[34] S. Li, D. Reddy, and B. Jacob, "A performance & power comparison of modern high-speed dram architectures," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18, Alexandria, Virginia, USA: Association for Computing Machinery, 2018, pp. 341–353, ISBN: 9781450364751.

[35] *Specifications - Supercomputer Fugaku : Fujitsu Global*, https://www.fujitsu.com/global/about/innovation/fugaku/specifications/, November 2020.

[36] *Summit - Oak Ridge Leadership Computing Facility*, https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[37] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, Austin, TX, USA: Association for Computing Machinery, 2009, pp. 267–278, ISBN: 9781605585260.

[38] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12, USA: IEEE Computer Society, 2012, pp. 1–12, ISBN: 9781467308274.

[39] *Gen-Z Consortium: Computer Industry Alliance Revolutionizing Data Access*, https://genzconsortium.org/.

[40] *Compute Express Link: The Breakthrough CPU-to-Device Interconnect*, https://www.computeexpresslink.org/.

[41] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity," Dec. 2018.

[42] K. Bergman, T. Conte, A. Gara, M. Gokhale, M. Heroux, P. Kogge, B. Lucas, S. Matsuoka, V. Sarkar, and O. Temam, "Future high performance computing capabilities: Summary report of the advanced scientific computing advisory committee (ascac) subcommittee," Mar. 2019.

[43] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, London, United Kingdom: Association for Computing Machinery, 2016, ISBN: 9781450342407.

[44] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, San Jose, California: Association for Computing Machinery, 2012, ISBN: 9781450317610.

[45] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18, Boston, MA, USA: USENIX Association, 2018, pp. 759–773, ISBN: 978-1-931971-44-7.

[46] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1304–1318, Apr. 2020.

[47] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable nvm," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19, Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 304–315, ISBN: 9781450372060.

[48] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, 2018, pp. 41–42, ISBN: 9781450356299.

[49] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[50] *PolyBench/C - The Polyhedral Benchmark suite*, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[51] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577, ISBN: 978-1-939133-08-3.

[52] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 13–24, ISBN: 9781450369732.

[53] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov, "Bandwidth-aware page placement in numa," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 546–556.

[54] *autonuma: Optimize memory placement for memory tiering system*, https://lwn.net/Articles/835402/, October 27, 2020.

[55] T. D. Doudali and A. Gavrilovska, "Mnemo: Boosting memory cost efficiency in hybrid memory systems," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 412–421.

[56] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[57] A. Mirhoseini, H. Pham, Q. Le, M. Norouzi, S. Bengio, B. Steiner, Y. Zhou, N. Kumar, R. Larsen, and J. Dean, "Device placement optimization with reinforcement learning," 2017.

[58] E. Ipek, O. Mutlu, J. F. Martınez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 39–50, Jun. 2008.

[59] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *Proceedings of the 35th In-*

ternational Conference on Machine Learning, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Oct. 2018, pp. 1919–1928.

[60] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. arXiv: 1412.6980.

[61] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[62] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015.

[63] *CORAL Benchmark Codes*, https://asc.llnl.gov/CORAL-benchmarks/, Dec. 2018.

[64] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Jan. 2011.

[65] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54, ISBN: 978-1-4244-5156-2.

[66] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 369–383, Mar. 2016.

[67] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris, "Shoal: Smart allocation and replication of memory for parallel programs," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA: USENIX Association, 2015, pp. 263–276, ISBN: 978-1-931971-225.

[68] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18, Boston, MA, USA: USENIX Association, 2018, pp. 951–965, ISBN: 9781931971447.

[69] T. D. Doudali, D. Zahka, and A. Gavrilovska, "The case for optimizing the frequency of periodic data movements over hybrid memory systems," in *The International Symposium on Memory Systems*, ser. MEMSYS 2020, Washington, DC, USA: Association for Computing Machinery, 2020, pp. 137–143, ISBN: 9781450388993.

[70] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly - Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 4, 2012.

[71] V. Deodhar, H. Parikh, A. Gavrilovska, and S. Pande, "Compiler Assisted Load Balancing on Large Clusters," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[72] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, "LoopProf : Dynamic Techniques for Loop Detection and Profiling," in *Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[73] *DynInst: Putting the Performance in High Performance Computing*, dyninst.org.

[74] H. A. Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 843–857, ISBN: 978-1-939133-14-4.

[75] N. N. Astakhova, L. A. Demidova, and E. V. Nikulchev, "Forecasting method for grouped time series with the use of k-means algorithm," *Applied Mathematical Sciences*, vol. 9, pp. 4813–4830, 2015.

[76] F. Martinez Alvarez, A. Troncoso, J. C. Riquelme, and J. S. Aguilar Ruiz, "Energy time series forecasting based on pattern sequence similarity," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 8, pp. 1230–1243, 2011.

[77] K. Bandara, C. Bergmeir, and S. Smyl, *Forecasting across time series databases using recurrent neural networks on groups of similar series: A clustering approach*, 2018. arXiv: 1710.03222 [cs.LG].

[78] Rui Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.

[79] *ImageNet*, https://image-net.org/.

[80] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 513–526, ISBN: 9781450371025.

[81] S. Byna, Y. Chen, and X.-H. Sun, "A taxonomy of data prefetching mechanisms," in *2008 International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008)*, 2008, pp. 19–24.

[82] *Interpolations for imshow - Matplotlib documentation*, https://matplotlib.org/stable/gallery/images_contours_and_fields/interpolation_methods.html.

[83] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in numa architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 369–380, ISBN: 9781450328098.

[84] T. Mason, T. D. Doudali, M. Seltzer, and A. Gavrilovska, "Unexpected performance of intel® optane™ dc persistent memory," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 55–58, 2020.

[85] T. D. Doudali and A. Gavrilovska, "Comerge: Toward efficient data placement in shared heterogeneous memory systems," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, Alexandria, Virginia: Association for Computing Machinery, 2017, pp. 251–261, ISBN: 9781450353359.

[86] L. Zhang, R. Karimi, I. Ahmad, and Y. Vigfusson, "Optimal data placement for heterogeneous cache, memory, and storage systems," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 1, May 2020.

[87] V. R. Kommareddy, A. Awad, C. Hughes, and S. D. Hammond, "Exploring allocation policies in disaggregated non-volatile memories," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, ser. MCHPC '18, Dallas, TX, USA: Association for Computing Machinery, 2018, pp. 58–66, ISBN: 9781450361132.

[88] C. Chou, A. Jaleel, and M. Qureshi, "Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, Alexandria, Virginia: Association for Computing Machinery, 2017, pp. 268–280, ISBN: 9781450353359.

[89] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 13–24.

[90] C. C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 1–12.

[91] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 861–873, ISBN: 9781450383172.

[92] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting memory accesses: The road to compact ml-driven prefetcher," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19, Washington, District of Columbia, USA: Association for Computing Machinery, 2019, pp. 461–470, ISBN: 9781450372060.

[93] A. Srivastava, T.-Y. Wang, P. Zhang, C. A. F. De Rose, R. Kannan, and V. K. Prasanna, "Memmap: Compact and generalizable meta-lstm models for memory access prediction," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2020, pp. 57–68.

[94] P. Zhang, A. Srivastava, B. Brooks, R. Kannan, and V. K. Prasanna, "Raop: Recurrent neural network augmented offset prefetcher," in *The International Symposium on Memory Systems*, ser. MEMSYS 2020, Washington, DC, USA: Association for Computing Machinery, 2020, pp. 352–362, ISBN: 9781450388993.

[95] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based memory allocation for c++ server workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 541–556, ISBN: 9781450371025.

[96] C. Mendis, A. Renda, D. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Sep. 2019, pp. 4505–4515.

[97] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 19–33, ISBN: 9781450362405.

[98] M. Maas, "A taxonomy of ml for systems problems," *IEEE Micro*, vol. 40, no. 5, pp. 8–16, 2020.

[99] L. Cen, R. Marcus, H. Mao, J. Gottschlich, M. Alizadeh, and T. Kraska, "Learned garbage collection," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2020, London, UK: Association for Computing Machinery, 2020, pp. 38–44, ISBN: 9781450379960.

[100] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," *SIGARCH Comput. Archit. News*, vol. 43, no. 3S, pp. 285–297, Jun. 2015.

[101] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, Beijing, China: Association for Computing Machinery, 2019, pp. 270–288, ISBN: 9781450359566.

[102] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, Jun. 2017, pp. 2430–2439.

[103] R. Chen, J. Wu, H. Shi, Y. Li, X. Liu, and G. Wang, "Drlpart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21, Virtual Event, Sweden: Association for Computing Machinery, 2020, pp. 175–188, ISBN: 9781450382175.

[104] S.-C. Kao, C.-H. H. Yang, P.-Y. Chen, X. Ma, and T. Krishna, "Reinforcement learning based interconnection routing for adaptive traffic optimization," in *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, ser. NOCS '19, New York, New York: Association for Computing Machinery, 2019, ISBN: 9781450367004.

[105] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 622–636.

[106] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 1–13, ISBN: 9781450366694.

[107] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: Association for Computing Machinery, 2018, pp. 489–504, ISBN: 9781450347037.

[108] *NVIDIA Deep Learning Accelerator*, http://nvdla.org/.

[109] *Intel Deep Learning Inference Accelerator*, https://www.intel.com/content/dam/support/us/en/documents/server-products/server-accessories/Intel_DLIA_UserGuide_1.0.pdf.

[110] *Intel FPGAS for Artificial Intelligence*, https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/overview.html.

[111] *Deep Learning Accelerators - Micron Technology, Inc.* https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/deep-learning-accelerators.

[112] *Cloud Tensor Processing Units (TPUs) - Google Cloud*, https://cloud.google.com/tpu/docs/tpus.

[113] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *SIGPLAN Not.*, vol. 53, no. 2, pp. 461–475, Mar. 2018.

[114] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[115] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 499–513, ISBN: 9781450362405.

[116] X. Wang, Y. Luo, D. Crankshaw, A. Tumanov, F. Yu, and J. E. Gonzalez, "IDK cascades: Fast deep learning by learning not to overthink," in *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, A. Globerson and R. Silva, Eds., AUAI Press, 2018, pp. 580–590.

[117] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased lstm: Accelerating recurrent network training for long or event-based sequences," in *Proceedings of the 30th Interna-*

tional Conference on Neural Information Processing Systems, ser. NIPS'16, Barcelona, Spain: Curran Associates Inc., 2016, pp. 3889–3897, ISBN: 9781510838819.

[118] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, Montréal, Canada: Curran Associates Inc., 2018, pp. 9031–9042.

[119] A. Mujika, F. Meier, and A. Steger, "Fast-slow recurrent neural networks," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017, pp. 5915–5924.

[120] J. Appleyard, T. Kocisky, and P. Blunsom, *Optimizing performance of recurrent neural networks on gpus*, 2016. arXiv: 1604.01946 [cs.LG].

[121] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *Cudnn: Efficient primitives for deep learning*, 2014. arXiv: 1410.0759 [cs.NE].

[122] *Amazon CloudWatch*, https://aws.amazon.com/cloudwatch/.

[123] *Grafana*, https://grafana.com/.

[124] *Nagios*, https://www.nagios.org/.

[125] *LLView - Graphical monitoring of batch system controlled cluster*, https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/LLview/_node.html.

[126] *Intel VTune Profiler*, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html.

[127] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara, "Kaleidoscope: Cloud micro-elasticity via vm state coloring," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, Salzburg, Austria: Association for Computing Machinery, 2011, pp. 273–286, ISBN: 9781450306348.

[128] M. Bojarski, D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *ArXiv*, vol. abs/1604.07316, 2016.

[129] N. Cohen, T. Balch, and M. Veloso, "Trading via image classification," *CoRR*, vol. abs/1907.10046, 2019. arXiv: 1907.10046.

[130] T. Estrada, J. Benson, H. Carrillo-Cabada, A. M. Razavi, M. A. Cuendet, H. Weinstein, E. Deelman, and M. Taufer, "Graphic encoding of macromolecules for efficient high-throughput analysis," in *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, ser. BCB '18, Washington, DC, USA: Association for Computing Machinery, 2018, pp. 315–324, ISBN: 9781450357944.

[131] H. Carrillo-Cabada, J. Benson, A. Razavi, B. Mulligan, M. A. Cuendet, H. Weinstein, M. Taufer, and T. Estrada, "A graphic encoding method for quantitative classification of protein structure and representation of conformational changes," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 1–1, 2019.

[132] K. Rong, C. E. Yoon, K. J. Bergen, H. Elezabi, P. Bailis, P. Levis, and G. C. Beroza, "Locality-sensitive hashing for earthquake detection: A case study of scaling data-driven science," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1674–1687, Jul. 2018.

[133] A. Matsunaga and J. A. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 495–504.

[134] *The CIFAR-10 dataset*, https://www.cs.toronto.edu/~kriz/cifar.html.

[135] L. Yu, J. Protze, O. Hernandez, and V. Sarkar, "Arbalest: Dynamic detection of data mapping issues in heterogeneous openmp applications," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 464–474.